Uniform Driver Interface

UDI Physical I/O Specification Version 1.01

http://www.project-UDI.org/specs.html

UDI Physical I/O Specification

Abstract

The UDI Physical I/O Specification defines the required interfaces and semantics for UDI environments that support Programmed I/O (PIO), Direct Memory Access (DMA), and Interrupts. This is an optional extension to the UDI Core Specification, which is defined in a separate book. The intended audience for this book includes driver writers, environment implementors, and metalanguage implementors.

UDI drivers that require physical I/O services must be written to conform to this specification, and can assume that all services described herein are available. Environments that don't need such drivers may choose not to support the physical I/O extensions, but any environment that supports UDI physical I/O drivers must conform to this specification, as well as to the UDI Core Specification.

Environments that support UDI drivers for devices that use any physical I/O bus types covered by other UDI specifications must support and conform to those Bus Binding specifications.

See the Document Organization chapter in the UDI Core Specification for a description of other books in the UDI Specification collection, as well as references to additional tutorial materials.

Status of This Document

This document has been reviewed by Project UDI Members and other interested parties and has been endorsed as a Final Specification. It is a stable document and may be used as reference material or cited as a normative reference from another document. This version of the specification is intended to be ready for use in product design and implementation. Every attempt has been made to ensure a consistent and implementable specification. Implementations should ensure compliance with this version.

Copyright Notice

Acknowledgements

The authors would like to thank everyone who reviewed working drafts of the specification and submitted suggestions and corrections.

The authors would especially like to thank their significant others for putting up with the many hours of overtime put into the development of this specification over long periods.

Thanks to the following folks who contributed significant amounts of time, ideas, or authoring in support of the development of this specification or in working on the prototype implementations which helped us validate the specification:

Richard Arndt (IBM)
Bob Barned (Lockheed Martin)
Mark Bradley (Adaptec)
Darren Busing (Adaptec)
Steve Bytnar (STG)
Thomas Clark (Sun)
Deven Corzine
Jack Craig (SCO)
Betty Dall (HP)
Tim Damron (IBM)
Burkhard Daniel (STG)
Don Dugger (Intel)
Mark Evenson (HP)
Barry Feild (SCO)
Scott Feldman (Intel)
Mike Firman (STG)
Kurt Gollhardt (SCO)
Bob Goudreau (Data General)
James Hall (SCO/Sun)
Jim Heidbrink (Lockheed Martin)
Chris Herzog (STG)
Chris Ilnicki (HP)
Bret Indrelee (SBS Technologies)
David Kahn (Sun)
Matt Kaufmann (SCO)
Andrew Knutsen (SCO)
Ahuva Kroizer (Intel)
Man Fai Lau (SCO)
John Lee (Sun)
Robert Lipe (SCO)
Mike Lyons (IBM)
Alex Malone (DEC)
Lynne McCue (IBM)
Bill Nicholls
Guru Pangal (Starcom)
Mark Parenti (DEC)
James Partridge (IBM)
Scott Popp (SCO)
Hiremane Radhakrishna (Intel)
John Ronciak (Intel)
Kevin Quick (Interphase)
Larry Robinson (Adaptec)
Andrew Schweig (STG)
Sam Shteingart (HP)
Ajmer Singh (SCO)
James Smart (Compaq)
Pete Smoot (HP)
David Stoft (HP)
Rob Tarte (Pacific Codeworks)
Wolfgang Thaler (Sun)
Ramaswamy Tummala (Starcom)
Linda Wang (Sun)
Kevin Van Maren (Unisys)
Mike Wenzel (HP)

Countless people have helped in one way or another and any omissions or errors on our part in the list above are just that: omissions or errors on our part.

Thanks to Kevin Quick and the folks at Interphase for hosting the Interoperability events which have provided a great venue for validating both prototype and production UDI products.

Finally, thanks to David Roberts (Certek Software Designs) for designing the Project UDI logo.

Table of Contents

List of Reference Pages by Chapter

Alphabetical List of Symbols

Alphabetical List of Symbols

UDI Physical I/O Specification

Section 1: Physical I/O Services

Introduction to Physical I/O 1

1.1 Overview

The UDI Physical I/O Specification specifies the services and metalanguage interfaces required to support a UDI physical I/O driver. A physical I/O driver is one that controls a directly-attached device (aka a *physical device*), which can be programmed without indirect access through another device. Physical devices typically interface to the host system via an I/O bridge (aka I/O bus) and have registers that are accessed by the host CPU via Programmed I/O (PIO). Physical devices may also generate interrupts to the I/O bridge and/or support direct access to system memory via the I/O bridge (aka DMA).

Section 1 introduces the general requirements, static driver property extensions, and additional UDI services required in order to support physical I/O. Section 2 defines the Bus Bridge Metalanguage, which is used by a physical I/O driver to communicate with its parent bus bridge driver. Section 3 defines the requirements for Bus Binding specifications.

1.2 General Requirements

Certain basic rules apply to all UDI Physical I/O drivers. In order to be UDI-compliant, a driver must follow all of these rules. UDI Physical I/O drivers must also follow the rules specified in the UDI Core Specification. Rules specific to Physical I/O drivers are listed here.

1.2.1 Versioning

All functions and structures defined in the UDI Physical I/O Specification, except for those defined in Chapter 5, *"Bus Bridge Metalanguage"*, are part of the "`udi_physio`" interface, currently at version "`0x101`". A driver that conforms to and uses the UDI Physical I/O Specification, Version 1.01, must include the following declaration in its `udiprops.txt` file (see Chapter 30, *"Static Driver Properties"*, of the UDI Core Specification):

```
requires udi_physio 0x101
```

In each UDI physical I/O driver source file, before including any UDI header files, the driver must define the preprocessor symbol, `UDI_PHYSIO_VERSION`, to indicate the version of the UDI Physical I/O Specification to which it conforms, which must be the same as the interface version defined above:

```
#define UDI_PHYSIO_VERSION 0x101
```

As defined in Section 30.4.6, "Requires Declaration," on page 30-6 of the UDI Core Specification, the two least-significant hexadecimal digits of the interface version represent the minor number; the rest of the hex digits represent the major number. Versions that have the same "major version number" as an earlier version shall be backward compatible with that earlier version (i.e. a strict superset).[1]

1. As an exception to this version compatibility, version 1.0 (0x100) is not forward compatible with any other versions bearing the major number of 1; version 1.0 of the specification cannot be wholly implemented as a functional product.

1.2.2 Header Files

Each device driver source file must include the file "`udi_physio.h`" after it includes "`udi.h`", as follows:

```
#include <udi.h>
#include <udi_physio.h>
```

These header files contain environment-specific definitions of standard UDI structures and types, as well as all function prototypes and other definitions needed to use the core and physical I/O UDI interfaces and services. Additional include files may be needed for other non-core services and metalanguages as defined in other UDI Specifications.

To maintain portability across UDI supportive platforms, device driver writers shall not assume any knowledge of the contents of these header files with respect to implementation-dependent aspects of the UDI interfaces (such as the definition of handles or abstract types). Similarly, drivers shall not access any functions or objects external to the driver except those defined in the UDI Specifications to which they conform.

1.2.3 Endianness Requirements

In general, UDI allows communicating drivers to run with different driver endianness (see Section 3.2.2, "Common Terms", in the UDI Core Specification), with the environment automatically handling endianness conversions across channels, except for the contents of UDI buffers (whose endianness is unknown to the environment). However, since the only mechanism for interrupt preprocessing (see **`udi_intr_attach_req`** on page 4-13) to pass significant amounts of data to a device driver's interrupt handler is via UDI buffers, and since this mechanism processes that data in the bus bridge driver's endianness, the bus bridge driver (which is the parent of the device driver) must be running with the same driver endianness as the device driver.

Therefore, any physical I/O driver shall be run with the same driver endianness as its parent.

1.3 Normative References

The UDI Physical I/O Specification references the UDI specifications listed below. These specifications contain provisions that, through reference in this document, constitute provisions of the UDI Physical I/O Specification.

1. UDI Core Specification, Version 1.01.

The UDI Physical I/O Specification also references the IEEE 1212.1 standard for Direct Memory Access Devices, but these references are informational and do not constitute provisions of the UDI Physical I/O Specification.

1.4 Extensions to Static Driver Properties

The UDI Physical I/O Specification defines the following extensions to the Static Driver Properties file, `udiprops.txt` (see Chapter 30, "*Static Driver Properties*" in the UDI Core Specification). These extensions are enabled by the presence of the following declaration anywhere in the file:

```
requires udi_physio 0x101
```

Any driver that uses functions or structures from the UDI Physical I/O Specification must include the above declaration in its Static Driver Properties.

The “region” declaration is extended to support the following additional region attributes:

Table 1-1 Physical I/O Region Attributes

`<region_attribute>`	`<value>`	**Meaning**
`type`	`interrupt`	An interrupt region. See **udi_intr_attach_req** on page 5-15.
`pio_probe`	`no`	Regions of this type never use `udi_pio_probe`. This is the default value for this attribute.
`pio_probe`	`yes`	Regions of this type may use `udi_pio_probe`. (See **udi_pio_probe** on page 4-30.)

A “nonsharable_interrupt” declaration is added:

```
nonsharable_interrupt <msgnum> [ <intr_idx> ]
```

At most one “nonsharable_interrupt” declaration must be included for each interrupt source of each supported device. If present, this indicates that the specified interrupt source does not support sharing, so must not be configured on the same interrupt line as another interrupt source. The `<msgnum>` must match `<msgnum>` in a “device” declaration, and indicates that this declaration applies to that device. The `<intr_idx>`, if present, selects one of several interrupt sources for the device, starting from zero; if not present, all interrupt sources for the device are affected. This declaration may indicate that the interrupt source is electrically non-sharable, or it may indicate a logically non-sharable interrupt source (one for which there is no way for the driver to determine if the interrupt has actually been asserted).

A “pio_serialization_limit” declaration is added:

```
pio_serialization_limit <max_idx>
```

The “pio_serialization_limit” declaration specifies the maximum ***`serialization_domain`*** index value that may be passed to `udi_pio_map` by the associated driver. A serialization domain is used to insure that all PIO handles defined for that domain will be serialized (i.e. execution of one PIO trans list must complete before execution of the next PIO trans list can be started). The driver developer must decide the number of serialization domains required for the implementation of the driver and declare that count in this declaration. Registering a PIO handle with a serialization domain index greater than the value specified in this declaration is a driver error (i.e. the value specified by the driver for the ***`serialization_domain`*** argument for the `udi_pio_map` operation must be in the range of 0..<`max_idx`>). If this declaration is not specified in the static properties file, the serialization limit defaults to zero (a single serialization domain) and all PIO handles (if any) must be registered with a ***`serialization_domain`*** index of 0. The maximum value for this declaration is 255.

1.5 Bus Bindings

The UDI Physical I/O Specification must be used in conjunction with one or more Bus Binding definitions for the type(s) of physical I/O bus on which a driver’s device is supported. A UDI Bus Binding definition provide the usage details that are specific to a particular bus type.

Chapter 6, “*Introduction to Bus Bindings*”, defines the general requirements on UDI Bus Binding definitions. The actual Bus Binding definitions are found in other specifications, such as the UDI PCI Bus Binding Specification.

DMA Constraints Management 2

2.1 Overview

The service calls in this chapter are used to manage *constraints objects* that are used to reflect drivers' constraints on buffer data and transfer properties. Constraints are used to indicate the data access capabilities and restrictions of the hardware and associated drivers as well as the various capabilities of software modules and drivers in the I/O handling path.

Constraints are first constructed in the environment by starting with a completely unconstrained set of constraints attributes, and are subjected to any restrictions imposed by the native bridge or any intervening bus bridges and then passed to the driver as its initial constraints object. Constraints may subsequently be set to more restrictive values by that driver or any other module in the data path, including the environment, the bus and any bus bridges, and any children of the driver.

To provide this capability, UDI defines a set of services used to set the values of various constraints. The environment implementation of constraints may include the ability to update the constraints at any time due to hardware events (*e.g.* hot swap), reconfiguration events (hardware or software), or changes in software parameters.

2.1.1 Constraints Attributes

Constraints may be imposed from several sources and are specified in terms of *constraints attributes.*

At the lowest level, constraints attributes indicate the host data access capabilities of the device. Any restrictions of the DMA engine on the device should be indicated by restricting one or more constraints attributes accordingly. For example, a device with only a 16-bit DMA address generation capability would indicate this by modifying the initial address-size constraints settings.

The constraints may then be combined in various combinations with other constraints or specific constraints attributes may be set by a driver to accurately represent the situations where the corresponding constraints apply. Drivers with different DMA constraints for different types of operations will typically have a different constraints handle for each operation type. Multiplexing drivers would typically combine constraints as dictated by the multiplexing possibilities to ensure that the constraints used are sufficient for all possible DMA uses.

The UDI environment need not adhere to constraints at resource allocation time. Instead, constraints may be ignored by the environment (if desired) up to such time as the constraint is applicable. A DMA constraint does not need to be explicitly adhered to until such time as a DMA mapping operation is performed. If the constraint is not honored at the time of initial resource allocation, the resource may need to be re-allocated at the time the constraint must be honored.

2.2 Constraints Attributes Service Calls and Structures

The functions in this section provide constraints attribute management services. These services include the ability to set or reset attributes on an existing constraints object, to combine constraints objects by merging attributes, and to make copies of existing constraints objects.

NAME

udi_dma_constraints_attr_spec_t *Specify attribute/value pair*

SYNOPSIS

```
#include <udi.h>

typedef struct {
     udi_dma_constraints_attr_t attr_type;
     udi_ubit32_t attr_value;
} udi_dma_constraints_attr_spec_t;
```

MEMBERS

attr_type is the attribute being specified.

attr_value is the value for the attribute.

DESCRIPTION

The `udi_dma_constraints_attr_spec_t` structure is used to associate an attribute with its corresponding value. An array of these structures is used as an argument for the `udi_dma_constraints_attr_set` operation to specify a list of attributes and their values to be set.

REFERENCES

`udi_dma_constraints_attr_t`, `udi_dma_constraints_attr_set`

NAME **udi_dma_constraints_attr_set** *Set constraints attributes*

SYNOPSIS

```
#include <udi.h>

void udi_dma_constraints_attr_set (
     udi_dma_constraints_attr_set_call_t *callback,
     udi_cb_t *gcb,
     udi_dma_constraints_t src_constraints,
     const udi_dma_constraints_attr_spec_t *attr_list,
     udi_ubit16_t list_length,
     udi_ubit8_t flags );

typedef void udi_dma_constraints_attr_set_call_t (
     udi_cb_t *gcb,
     udi_dma_constraints_t new_constraints,
     udi_status_t status );

/* Constraints Flags */
#define UDI_DMA_CONSTRAINTS_COPY          (1U<<0)
```

ARGUMENTS

callback, gcb are standard arguments described in the "Asynchronous Service Calls" section of *"Standard Calling Sequences"* in the UDI Core Specification

src_constraints is a constraints handle for the constraints object to be modified.

attr_list is the list of attributes and values to set.

list_length is the number of valid entries in the ***attr_list*** array.

flags is a bitmask of optional flags, which may include zero or more of the following:

`UDI_DMA_CONSTRAINTS_COPY` Make a copy of ***src_constraints*** before applying the new attributes.

new_constraints is a new constraints handle for the modified constraints object. This replaces the ***src_constraints*** handle.

status is a UDI status code indicating the success or failure of the constraints modification operation.

DESCRIPTION

`udi_dma_constraints_attr_set` sets or changes one or more attribute values in the constraints object referenced by the ***src_constraints*** handle. The ***attr_list*** argument indicates this driver's requirements for the specified constraints attributes.

If `UDI_DMA_CONSTRAINTS_COPY` is set in ***flags***, then ***src_constraints*** must be non-null, and a new constraints object will be allocated with its values copied from ***src_constraints***; the new attribute values from ***attr_list*** will be applied to the new copy, which will then be passed back to the driver via ***new_constraints***; the original source constraints object will retain its original values, but must not again be referenced by the driver until the callback.

If UDI_DMA_CONSTRAINTS_COPY is not set, the original source object (if any) will be consumed, and a handle to the newly modified object will be passed back to the driver via ***new_constraints*** in the callback. In this case, the driver must subsequently use the ***new_constraints*** value in place of ***src_constraints***. If ***src_constraints*** was null, a new constraints object will be allocated.

Most attributes have a defined range of least restrictive to most restrictive values, as specified with the description of the attribute. In these cases, udi_dma_constraints_attr_set sets the new value for an attribute to be the *more* restrictive of the specified ***attr_list*** entry's attr_value and the attribute's current value.

For example, if the current value of UDI_DMA_ELEMENT_LENGTH_BITS were 64 and the ***attr_list*** entry's attr_value were 32, the new value would be 32; but if ***attr_list*** entry's attr_value were 128, the attribute value would remain set to 64, since 64 is more restrictive than 128 for UDI_DMA_ELEMENT_LENGTH_BITS.

A few attributes have no defined sense of less or more restrictive. These are marked as "N/A" (not applicable). For these attributes, the new value is simply set to the value specified in the ***attr_list*** entry.

Once all attributes in the ***attr_list*** have been processed, the ***callback*** routine is invoked to notify the driver that the constraints object has been modified and the success or failure of that modification.

The udi_dma_constraints_attr_set operation may fail with ***status*** set to UDI_STAT_NOT_SUPPORTED if the requested attributes cannot be supported on a given system. The UDI_STAT_NOT_SUPPORTED error case is not intended to catch invalid attribute values. If attribute values are used that are outside the valid ranges documented for the attribute, the results are implementation-dependent.

If ***status*** indicates failure, the constraints object passed back in ***new_constraints*** shall have the same constraints attribute values as the original, and no additional constraints objects shall be allocated (even if UDI_DMA_CONSTRAINTS_COPY was set in the call).

WARNINGS

Control block usage must follow the rules described in the "Asynchronous Service Calls" section of *"Standard Calling Sequences"* in the UDI Core Specification.

Use of the ***attr_list*** parameter must conform to the rules described in Section 5.2.1.1, "Using Memory Pointers with Asynchronous Service Calls," on page 5-2.

STATUS VALUES

UDI_OK	successful modification of the constraints attributes

UDI_STAT_NOT_SUPPORTED environment and/or platform cannot support the requested combination of attributes.

REFERENCES `udi_dma_constraints_attr_spec_t`, `udi_dma_constraints_attr_reset`

NAME

udi_dma_constraints_attr_reset *Reset a constraints attribute to default*

SYNOPSIS

```
#include <udi.h>

void udi_dma_constraints_attr_reset (
     udi_dma_constraints_t constraints,
     udi_dma_constraints_attr_t attr_type );
```

ARGUMENTS

constraints is a constraints handle for the constraints object to be modified.

attr_type is the attribute to reset.

DESCRIPTION

`udi_dma_constraints_attr_reset` is used to reset a constraints attribute back to its default value (which is also usually the least restrictive). This is usually needed when a particular module provides special handling relative to the constraints attribute such that any restrictions imposed by parent or child drivers are not transferred through this driver.

REFERENCES

`udi_dma_constraints_attr_set`, `udi_dma_constraints_attr_t`

NAME **udi_dma_constraints_free** *Free a constraints object*

SYNOPSIS

```
#include <udi.h>

void udi_dma_constraints_free (
     udi_dma_constraints_t constraints );
```

ARGUMENTS

constraints is a handle for the constraints object being deallocated.

DESCRIPTION

udi_dma_constraints_free releases all resources associated with the constraints object.

If ***constraints*** is equal to UDI_NULL_CONSTRAINTS, explicitly or implicitly (zeroed by initial value or by using udi_memset), this function acts as a no-op. Otherwise, ***constraints*** must have been allocated by udi_dma_constraints_combine or passed to the driver via a channel operation.

Direct Memory Access (DMA) 3

3.1 Overview

Direct Memory Access (DMA) refers to data transfers initiated by a device to access system memory. This is contrasted with Programmed I/O (PIO), which involves transfers initiated by a CPU under driver software control to access registers or memory on a device.

UDI DMA services can be used to set up a UDI buffer or a piece of system memory for a DMA data transfer, to build scatter/gather lists, and to synchronize DMA caches. UDI DMA service calls use an opaque DMA handle to refer to DMA resources managed by the environment. DMA handles are allocated by `udi_dma_prepare` or `udi_dma_mem_alloc` and use the `udi_dma_handle_t` opaque handle type.

In order for a buffer or system memory to be usable for DMA, it must first be mapped for DMA access and a scatter/gather list describing the relevant address ranges must be built. This is accomplished using `udi_dma_buf_map` for UDI buffers or `udi_dma_mem_alloc` for shared control structures in system memory. These functions provide the driver with an IEEE 1212.1 compatible DMA scatter/gather list (see “DMA Block Vector Segment” below). The scatter/gather list contains a set of bus address-length pairs, referencing the DMA target, which the driver may use as-is, translate in place, or copy to the scatter/gather entries of its device’s DMA engine, as is appropriate for its device.

If a buffer is too big to handle in one DMA transfer (for example, due to constraints on the number of scatter/gather elements that the DMA engine can support), `udi_dma_buf_map` will set up a partial transfer. The driver will then perform multiple transfers in order to complete the original buffer request, calling `udi_dma_buf_map` repeatedly to refill the scatter/gather list after each partial transfer.

3.2 DMA Block Vector Segment

As mentioned in the UDI Core Specification, Chapter 13, *“Buffer Management”*, buffers are logically contiguous but possibly virtually and physically scattered. Such discontiguities are not visible to drivers, however, except in the way they are mapped for DMA so they can be accessed by a driver’s device. The data structures in this section are used to represent the individual address-length pairs for each contiguous (from the card’s viewpoint) block of data. These structures are intended to be as simple and convenient as possible for communicating between UDI and the driver. However, when mapping buffers for DMA, it would be even more advantageous if the I/O card could traverse these structures directly. To this end, UDI uses IEEE-1212.1 compatible scatter/gather structures.

I/O Block Vector Segments are IEEE-1212.1 compatible data structures that allow the I/O Unit to perform reads (writes) to (from) more than one physical segment (e.g., page) in a single request. The IEEE-1212.1 DMA Framework defines standard structures for representing physically-scattered but logically-contiguous data. There are two formats defined, one for buses with 32-bit addresses and one for 64-bit addresses, as shown in Figure 3-1.

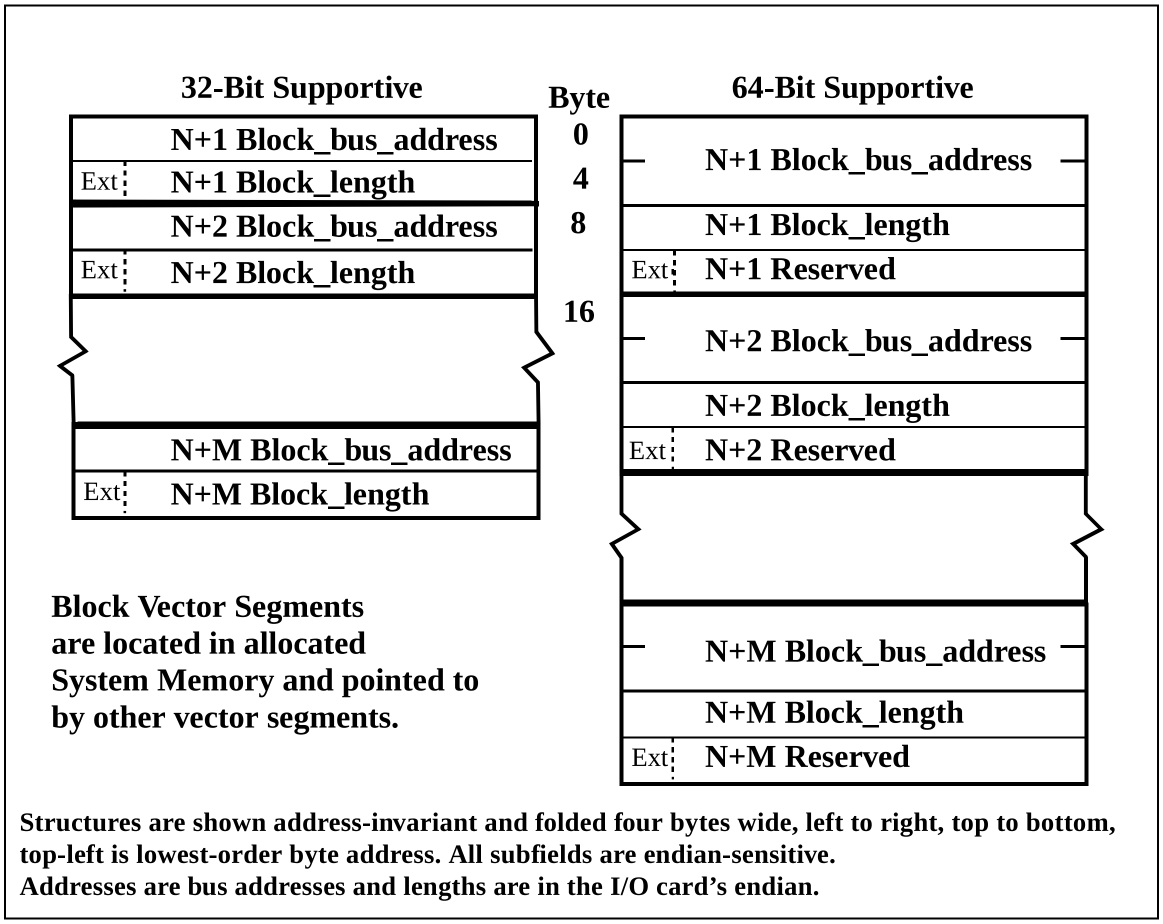
32-Bit Supportive
Byte
64-Bit Supportive
N+1 Block_bus_address
Ext
N+1 Block_length
N+2 Block_bus_address
N+2 Block_length
N+M Block_bus_address
N+M Block_length
0
4
8
16
N+1 Reserved
N+2 Reserved
N+M Reserved
Block Vector Segments are located in allocated System Memory and pointed to by other vector segments.
Structures are shown address-invariant and folded four bytes wide, left to right, top to bottom, top-left is lowest-order byte address. All subfields are endian-sensitive.
Addresses are bus addresses and lengths are in the I/O card's endian.

Figure 3-1 IEEE-1212.1 Block Vector Structure

As described in IEEE-1212.1, each address-length pair in a block vector segment points to and delimits a block of data bytes that is contiguous from the perspective of the I/O card. If the *Ext* bit is 0 for a pair, the address references payload data. If the *Ext* bit is 1, the address references an extension of the Block Vector structure itself (i.e., another array of address-length pairs) and the length indicates the physical size of this extension segment in bytes (which facilitates pre-fetch). Each block vector segment can contain at most one extension address; trees of vector segments are not allowed but chains of segments (end-to-start) are allowed. Any address-length pairs that may follow an extension address are ignored.

UDI limits its use of the flexibility allowed by IEEE-1212.1 in that for any Block Vector segment created by the environment, only the last address-length pair in a segment will ever be an indirect pointer to another Block Vector segment. All other address-length pairs will only reference data. This allows drivers for devices that are only partially IEEE-1212.1 compatible to manipulate address-length pairs without searching for indirect Block Vector references.

Environments must construct block vector segments according to the following rules (adapted from IEEE-1212.1) and the driver-specified DMA constraints (see **udi_dma_constraints_attr_t** on page 3-15).

1. Environments should use as few segments as possible, for space and time efficiency.

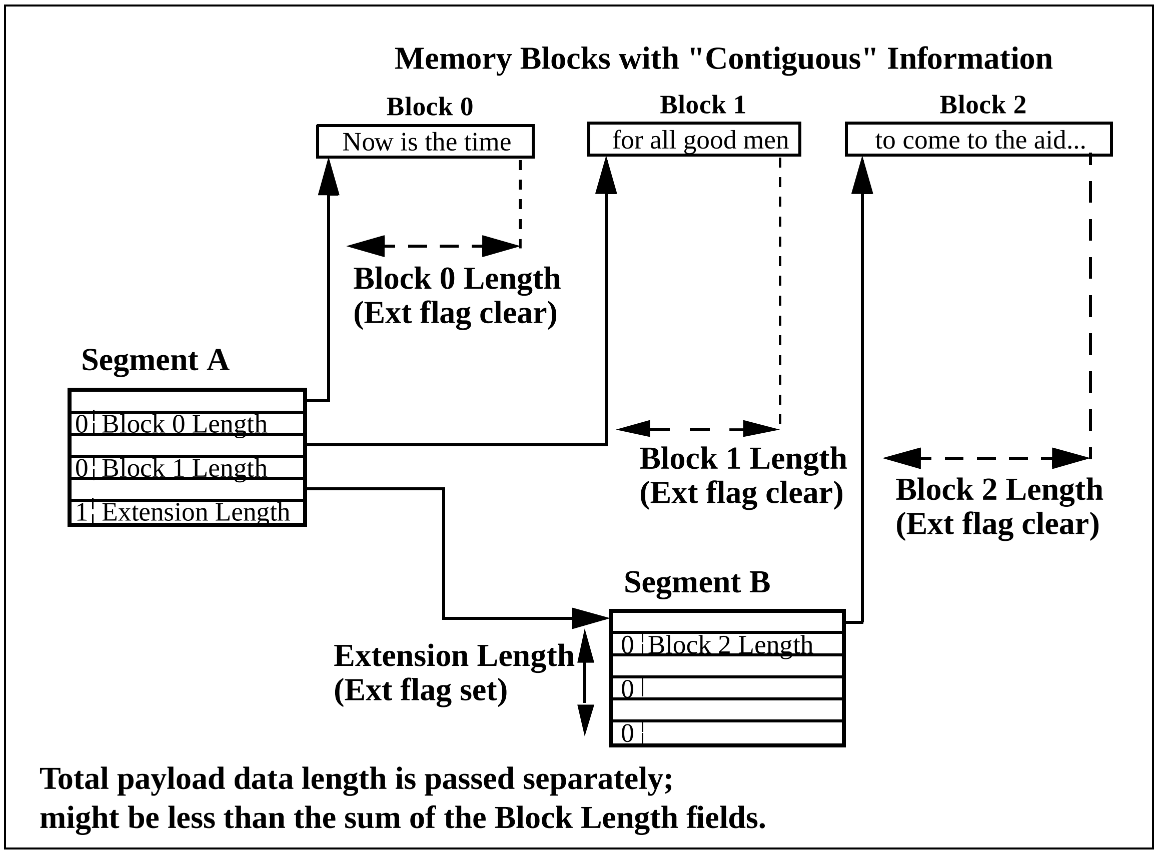

Figure 3-2 Example of Vector Use and Extension

2. When extension segments are used, the UDI_SCGTH_EXT flag bit shows whether an element references payload data (0) or the next Block Vector extension segment (1). If an extension segment is referenced, the ***block_length*** field gives the size of the next segment (making it easier for the I/O Unit to handle the segment in one block-copy operation, if desired); this size includes the indirect element, if any. The length of the last segment may extend beyond the end of the last element. The driver (or device) can always realize it has reached the last element in a segment by either encountering an indirect element, or by reaching the required total data length or total number of direct elements (see ***scgth_num_elements*** in **udi_scgth_t** on page 3-10).

3. When a data segment is referenced by ***block_busaddr*** (i.e., UDI_SCGTH_EXT is 0), the byte referenced by this I/O address will be aligned as specified by the associated DMA constraints; the ***block_length*** field, which may also be odd and will be ≥ zero, gives the number of contiguous bytes of data to be sent, or space available to receive data.

4. The first indirect element in a vector segment causes the I/O Unit to follow to the next segment, ignoring any following address-length pairs in the current Block Vector segment. Therefore, for speed and simplicity, Block Vector Segments are single-threaded rather than stacked to build trees, arrays, or circular lists of extension segments. Only the last address-length pair in a segment will ever contain an indirect element; the last address-length pair in a Block Vector segment will only contain an indirect element, or be unused at the end of the data transfer.

5. The total amount of data contained by the logically contiguous buffer represented by these structures is carried by some other vehicle (normally a metalanguage control block) and a transfer must be terminated when this number of bytes have been handled, regardless of whether additional bytes of buffer space are indicated by the address-length pairs. This is true for both inbound and outbound data transfers. All elements in a vector are usable.
6. Null address values are not defined since I/O bus addresses are used and all values are valid addresses. Zero-length data segments are allowed for flexibility in use by higher software layers (however, they are not encouraged because of the slight performance impact). These shall be ignored. ***Block_length*** must not be zero if the UDI_SCGTH_EXT flag is set.
7. All address-length pairs of a given block vector must have the same structure, 32- versus 64-bit supportive (therefore, one ***scgth_format*** value applies to the entire scatter/gather list). Also, for card/host compatibility, block vector segments will be aligned according to the address size of their format; i.e., 32-bit supportive block vector segments shall be at least 4-byte aligned, and 64-bit supportive segments shall be at least 8-byte aligned.
8. All fields of block vector segments are endian-sensitive, including the UDI_SCGTH_EXT flag.
9. The reserved field for the 64-bit format shall be set to zero when initialized and must be ignored when read.

3.3 DMA Service Calls and Structures

NAME

udi_dma_handle_t *DMA handle type*

SYNOPSIS

```
#include <udi.h>
#include <udi_physio.h>

typedef <HANDLE> udi_dma_handle_t;

/* Null handle value for udi_dma_handle_t */
#define UDI_NULL_DMA_HANDLE         <NULL_HANDLE>
```

DESCRIPTION

The DMA handle type, `udi_dma_handle_t`, holds an opaque handle that refers to DMA resources managed by the environment.

DMA handles are not transferable between regions.

NAME

udi_dma_limits *Platform-specific allocation and access limits*

SYNOPSIS

```
#include <udi.h>
#include <udi_physio.h>

typedef struct {
     udi_size_t max_legal_contig_alloc;
     udi_size_t max_safe_contig_alloc;
     udi_size_t cache_line_size;
} udi_dma_limits_t;

void udi_dma_limits (
     udi_dma_limits_t *dma_limits );

/* Guaranteed Minimum Allocation Size */
#define UDI_DMA_MIN_ALLOC_LIMIT           4000
```

MEMBERS

max_legal_contig_alloc is the maximum legal size of a single DMA-able memory element (one element in a `udi_scgth_t` scatter/gather list) that can be requested from `udi_dma_mem_alloc` or `udi_dma_buf_map`. Any request for a larger size will produce indeterminate results, which could include termination of the driver or region, or even a complete system abort.

max_safe_contig_alloc is the maximum size of a single DMA-able memory element that should be requested from `udi_dma_mem_alloc` or `udi_dma_buf_map` without being prepared to cancel an unsuccessful allocation.

cache_line_size is the size, in bytes, of the largest cache line that affects DMA-able memory.

DESCRIPTION

`udi_dma_limits_t` reflects the DMA memory allocation limits available on a particular system, for a particular region. These limits may vary from region to region, but will remain constant for the life of a region.

The `udi_dma_limits_t` structure is passed back to a driver by a call to `udi_dma_limits`.

Since UDI can be implemented on a wide variety of systems from small embedded systems to large server systems, the ability to provide contiguous DMA-able memory through `udi_dma_mem_alloc` and `udi_dma_buf_map` can vary widely. `udi_dma_limits` allows drivers to adjust their allocation algorithms to best fit their environment.

There are two types of allocation limits: *legal* limits and *safe* limits. Legal limits represent the absolute upper bound on a single allocation. Drivers must not make requests that would exceed the legal limits.

Safe limits represent the maximum amount that a driver may safely request without arranging to deal with unsuccessful allocations. For any size greater than the safe limit (but not exceeding the legal limit), drivers must cancel the request (using `udi_cancel`) after a reasonable amount of time has expired. To do this, they must set a timer using `udi_timer_start` or `udi_timer_start_repeating`. Drivers may also cancel allocations below the safe limit, but they are not expected to do so.

The ***max_legal_contig_alloc*** and ***max_safe_contig_alloc*** limits affect the size of a DMA-able memory element (one element in a `udi_scgth_t` scatter/gather list), which must be contiguous in bus address space. In most cases, DMA constraints allow the environment sufficient flexibility to use smaller pieces; in such cases, these limits wouldn't matter. However, the following DMA and transfer constraints attributes could force the environment to use larger element sizes:

UDI_DMA_ELEMENT_GRANULARITY_BITS

UDI_DMA_SCGTH_MAX_ELEMENTS

UDI_DMA_NO_PARTIAL

UDI_XFER_GRANULARITY

All of the above allocation limits are guaranteed to be greater than or equal to **UDI_DMA_MIN_ALLOC_LIMIT** (4000 bytes). This means drivers don't need to check these limits for requests that don't exceed 4000 bytes.

The ***cache_line_size*** value may be used to set appropriate DMA constraints for devices that need data or "slop" aligned on cache line boundaries.

REFERENCES `udi_limits_t, udi_dma_mem_alloc, udi_dma_buf_map, udi_cancel`

NAME **udi_busaddr64_t** *64-bit bus address data type*

SYNOPSIS

```
#include <udi.h>
#include <udi_physio.h>

typedef <OPAQUE> udi_busaddr64_t;
```

DESCRIPTION

The `udi_busaddr64_t` type used for bus addresses in the UDI_SCGTH_64 format is a self-contained opaque type that is guaranteed to be 64 bits in size and hold a bus address in the appropriate endianness. Drivers may copy `udi_busaddr64_t` values using assignment statements, or pass them by value or by reference to function calls, but may not use arithmetic operations or otherwise make assumptions about the internal structure of this data type. When swapping endianness of a `udi_busaddr64_t` value, drivers must use an 8-byte (64-bit) transaction size with a utility function to ensure proper endianness conversion (see Section 22.2.3, “Endian-Swapping Utilities,” on page 22-11 of the UDI Core Specification).

WARNINGS

The `udi_busaddr64_t` type is not transferable between regions.

NAME **udi_scgth_t** *I/O Bus scatter/gather structure*

SYNOPSIS

```
#include <udi.h>
#include <udi_physio.h>

typedef struct {
     udi_ubit32_t block_busaddr;
     udi_ubit32_t block_length;
} udi_scgth_element_32_t;

typedef struct {
     udi_busaddr64_t block_busaddr;
     udi_ubit32_t block_length;
     udi_ubit32_t el_reserved;
} udi_scgth_element_64_t;

/* Extension Flag */
#define UDI_SCGTH_EXT                        0x80000000

typedef struct {
     udi_ubit16_t scgth_num_elements;
     udi_ubit8_t scgth_format;
     udi_boolean_t scgth_must_swap;
     union {
          udi_scgth_element_32_t *el32p;
          udi_scgth_element_64_t *el64p;
     } scgth_elements;
     union {
          udi_scgth_element_32_t el32;
          udi_scgth_element_64_t el64;
     } scgth_first_segment;
} udi_scgth_t;

/* Values for scgth_format */
#define UDI_SCGTH_32                         (1U<<0)
#define UDI_SCGTH_64                         (1U<<1)
#define UDI_SCGTH_DMA_MAPPED                 (1U<<6)
#define UDI_SCGTH_DRIVER_MAPPED              (1U<<7)
```

MEMBERS

block_busaddr is the I/O card-relative bus address of a contiguous block of data bytes (if `UDI_SCGTH_EXT` bit is cleared), or an extension array of `udi_scgth_element_t` elements (if `UDI_SCGTH_EXT` is set). These blocks are contiguous from the I/O card's perspective. See the **DESCRIPTION** below for comments on device endianness.

block_length is the length of the related block. If this is an extension element (the `UDI_SCGTH_EXT` bit is set) then this is the size in bytes of the next scatter/gather block, including the indirect element in that block, if present. If this is not an extension element then this is the size in bytes of the DMA data associated with this ***block_busaddr***. In the `UDI_SCGTH_32` format, the `UDI_SCGTH_EXT` bit is part of this field.

el_reserved includes the `UDI_SCGTH_EXT` bit in the `UDI_SCGTH_64` format. Other bits are unused.

UDI_SCGTH_EXT is a mask value, to be applied to the ***block_length*** field of a `udi_scgth_element_32_t` or the ***el_reserved*** field of a `udi_scgth_element_64_t`. If this bit is set, then the scatter/gather element is an indirect reference to a possibly physically-discontiguous continuation of the scatter/gather list.

scgth_num_elements gives the number of direct `udi_scgth_element_t` elements included in the entire scatter/gather list (indirect elements are not included, but elements from all segments are included). This may be used to estimate I/O driver control resources.

scgth_format indicates the format of the scatter/gather elements. Exactly one of the following flags will be set in ***scgth_format***, corresponding to the 32-bit and 64-bit scatter/gather formats, respectively:
UDI_SCGTH_32
UDI_SCGTH_64

In addition, the **UDI_SCGTH_DMA_MAPPED** flag will be set if the scatter/gather list is “DMA-mapped”, meaning that the list elements themselves are made readable from the adapter via DMA and are in an endianness appropriate for access by the device (as indicated by the UDI_DMA_SCGTH_ENDIANNESS constraints attribute). The driver must ensure that its DMA device does not write to the scatter/gather list memory.

The **UDI_SCGTH_DRIVER_MAPPED** flag will be set if the scatter/gather list is (also) “driver-mapped”, meaning that the list elements are readable from the driver via the ***el32p*** or ***el64p*** pointer. Otherwise, these pointers are unused and their value is unspecified. The list elements will be in the driver’s endianness if and only if driver-mapped and not DMA-mapped.

The ***scgth_format*** value used for a scatter/gather list is determined by the value of the UDI_DMA_SCGTH_FORMAT constraints attribute used to create the scatter/gather list.

scgth_must_swap is a flag indicating that the driver must swap endianness when accessing driver-mapped list elements via ***el32p*** or ***el64p***. This flag will always be `FALSE` if the scatter/gather list is not DMA-mapped. If not driver-mapped, the value of the flag is unspecified and must not be used. When the scatter/gather list is both DMA-mapped and driver-mapped, the driver must check this flag; it must not assume it knows whether or not to swap, based on its specified `UDI_DMA_SCGTH_ENDIANNESS`, since interposed bus bridges may change the endianness.

See Section 22.2, "Endianness Management," on page 22-2 of the UDI Core Specification for details on how to construct C structure definitions for proper endianness handling, and the endian swapping utilities that are available.

el32p, el64p are members of the ***scgth_elements*** union. One of these, depending on the value of ***scgth_format***, will be set to be a pointer to the scatter/gather list in the driver's address space if UDI_SCGTH_DRIVER_MAPPED is set in ***scgth_format***.

scgth_first_segment is only used for DMA-mapped scatter/gather lists. Depending on the value of ***scgth_format***, ***el32*** or ***el64*** contains the I/O-card relative address of the beginning of the scatter/gather list and the length in bytes of the first segment in this list. ***scgth_first_segment*** is similar to an IEEE-1212.1 indirect element, except that the UDI_SCGTH_EXT bit is never set.

DESCRIPTION

The `udi_scgth_t` structure provides a way to access the block vector array segment(s) that describe the memory addresses of a DMA-mapped data buffer or shared control structure memory. Scatter/gather lists are built by the environment as a result of calls to `udi_dma_buf_map` or `udi_dma_mem_alloc`.

Scatter/gather lists are presented in either 32-bit or 64-bit format, as requested by the driver through its DMA constraints. The format used is reported back to the driver through ***scgth_format***. (See UDI_DMA_SCGTH_FORMAT on page page 3-18 in **udi_dma_constraints_attr_t** on page 3-15.)

Depending on the value of the UDI_DMA_SCGTH_FORMAT constraints attribute, the scatter/gather elements may be DMA-mapped, driver-mapped, or both. The driver chooses the appropriate visibility depending on how closely its device's scatter/gather format resembles the UDI scatter/gather list format (which is based on IEEE-1212.1). If they are identical, the driver should choose UDI_SCGTH_DMA_MAPPED and let its device access the scatter/gather list directly. If the formats are similar, but not identical, the driver may choose to use both UDI_SCGTH_DMA_MAPPED and UDI_SCGTH_DRIVER_MAPPED and modify the scatter/gather elements in place (being careful to observe ***scgth_must_swap*** if set) before having the device access the list. Otherwise, the driver should use just UDI_SCGTH_DRIVER_MAPPED and use separately allocated DMA-able memory or PIO to build a scatter/gather list in the appropriate format, based on the values it reads from the ***scgth_elements*** array.

If DMA-mapped, the memory for the scatter/gather segments themselves will conform to the DMA constraints related to scatter-gatter memory, and the associated `udi_scgth_t` structure will contain an indirect reference to the first element of the scatter/gather list in ***scgth_first_segment***. The members of the ***el32*** or ***el64*** element within this union are in the driver's endianness. The ***block_busaddr*** member contains the I/O card-relative address of the beginning of the scatter/gather list, and the ***block_length*** member contains the length of the first segment of this list, including an indirect element, if any exists.

If not DMA-mapped, the ***scgth_first_segment*** field of `udi_scgth_t` is not valid, and the scatter/gather list will not contain extension elements (UDI_SCGTH_EXT); the entire list is provided in a single block vector segment.

The environment's use of the IEEE-1212.1 structures is such that all the `udi_scgth_element_t` elements in a block vector segment are used before chaining to the next segment. Therefore, the driver can use the length of the block vector segment to determine where the next indirect `udi_scgth_element_t` pair is, if any. Therefore, it is not necessary for the driver to check for UDI_SCGTH_EXT for other elements in the segment. Note, though, that the last segment may contain fewer elements than covered by the length of the segment; the total number of direct elements (***scgth_num_elements***) or the total size of the mapped memory must be used to determine that the end of the list has been reached.

WARNINGS

The `udi_scgth_t` type is not transferable between regions.

NAME

udi_dma_constraints_t *UDI DMA constraints handle*

SYNOPSIS

```
#include <udi.h>
#include <udi_physio.h>

typedef <HANDLE> udi_dma_constraints_t;

/* NULL dma constraints constant */
#define UDI_NULL_DMA_CONSTRAINTS       <NULL_HANDLE>
```

DESCRIPTION

When buffer data is passed to a driver, it may be required to satisfy a set of DMA constraints, such as maximum transfer size or device offset alignment. In order for drivers to express their DMA constraints UDI defines a DMA constraints object, which is accessed via an opaque *dma constraints handle*:

DMA Constraints handles are used by the environment to ensure that when a buffer or DMA memory is allocated by the parent, the environment can optimize the allocation appropriately to meet all the constraints along the path, potentially avoiding copies later. The environment may also add its own hidden constraints based on knowledge of how the buffer or DMA memory will be used in the rest of the operating system.

Constraints handles are transferable between regions.

WARNINGS

Drivers must not compare handle values for equality, but the `UDI_HANDLE_IS_NULL` macro can be used to determine if a handle variable currently holds a null value.

REFERENCES

`UDI_HANDLE_IS_NULL`

NAME **udi_dma_constraints_attr_t** *DMA constraints attributes*

SYNOPSIS

```
#include <udi.h>
#include <udi_physio.h>

typedef udi_ubit8_t udi_dma_constraints_attr_t;

/* DMA Convenience Attribute Codes */
#define UDI_DMA_ADDRESSABLE_BITS          100
#define UDI_DMA_ALIGNMENT_BITS            101

/* DMA Constraints on the Entire Transfer */
#define UDI_DMA_DATA_ADDRESSABLE_BITS     110
#define UDI_DMA_NO_PARTIAL                111

/* DMA Constraints on the Scatter/Gather List */
#define UDI_DMA_SCGTH_MAX_ELEMENTS        120
#define UDI_DMA_SCGTH_FORMAT              121
#define UDI_DMA_SCGTH_ENDIANNESS          122
#define UDI_DMA_SCGTH_ADDRESSABLE_BITS    123
#define UDI_DMA_SCGTH_MAX_SEGMENTS        124

/* DMA Constraints on Scatter/Gather Segments */
#define UDI_DMA_SCGTH_ALIGNMENT_BITS      130
#define UDI_DMA_SCGTH_MAX_EL_PER_SEG      131
#define UDI_DMA_SCGTH_PREFIX_BYTES        132

/* DMA Constraints on Scatter/Gather Elements */
#define UDI_DMA_ELEMENT_ALIGNMENT_BITS    140
#define UDI_DMA_ELEMENT_LENGTH_BITS       141
#define UDI_DMA_ELEMENT_GRANULARITY_BITS  142

/* DMA Constraints for Special Addressing */
#define UDI_DMA_ADDR_FIXED_BITS           150
#define UDI_DMA_ADDR_FIXED_TYPE           151
#define UDI_DMA_ADDR_FIXED_VALUE_LO       152
#define UDI_DMA_ADDR_FIXED_VALUE_HI       153

/* DMA Constraints on DMA Access Behavior */
#define UDI_DMA_SEQUENTIAL                160
#define UDI_DMA_SLOP_IN_BITS              161
#define UDI_DMA_SLOP_OUT_BITS             162
#define UDI_DMA_SLOP_OUT_EXTRA            163
#define UDI_DMA_SLOP_BARRIER_BITS         164

/* Values for UDI_DMA_SCGTH_ENDIANNESS */
#define UDI_DMA_LITTLE_ENDIAN             (1U<<6)
#define UDI_DMA_BIG_ENDIAN                (1U<<5)

/* Values for UDI_DMA_ADDR_FIXED_TYPE */
#define UDI_DMA_FIXED_ELEMENT             1
```

```
#define UDI_DMA_FIXED_LIST              2
#define UDI_DMA_FIXED_VALUE             3
```

DESCRIPTION

This type is used to select a particular attribute of a constraints object. Constrants objects, referenced by constraints handles (see **udi_dma_constraints_t** on page 3-14), are used to constrain data and transfer properties to most optimally meet the requirements of all drivers handling the data.

A constraints attribute is a mnemonic constant (UDI_DMA_XXX in the tables below) which is used to set an associated value in a constraints handle. Definitions of use and the meaning of each attribute follow the tables.

There are two general types of constraints: restrictions and capabilities. Restriction constraints indicate that the operation is restricted to a specific value or set of values, and if propagated, restriction constraints typically become more constrictive with each module. For these types of constraints, there is a default value along with a minimum and maximum value (specified in the tables below or elsewhere) and the effects of the combine operation are to make the constraint the more or less restrictive value of the attributes being combined. The tables also specify any special interpretation for a constraint vaue of zero or N/A if zero is not a special case.

The capability constraints are used to indicate the various capabilities of a module and are not as simply described as the restriction constraints. Capability constraints are not usually linear values and don’t have a standard meaning when used with combine operations. For these types of constraints, the most and least restrictive values are specified as “N/A” indicating that they do not apply and the description of the constraint must be consulted to determine the effects of the combine operation.

Constraints objects, referenced by constraints handles (`udi_dma_constraints_t`), are used to constrain transfer properties as well as memory placement of `udi_buf_t` data, memory allocated by `udi_dma_mem_alloc`, and scatter/gather lists, in order to make them addressable via DMA from a particular device.

Any driver that will be using DMA, or drives a bus bridge that has an effect on DMA transfers, must apply its DMA constraints attributes to a constraints object by calling `udi_dma_constraints_attr_set`. The driver must then pass the constraints handle to `udi_dma_prepare` or `udi_dma_mem_alloc` to allow the environment to allocate and/or copy the memory and mapping registers used for DMA so that they meet the specified constraints. If different DMA transactions have different alignment restrictions (e.g., transmit DMA can be byte-aligned, but receive DMA must be 4-byte aligned), the driver would need to create a separate handle for each case.

A list of supported DMA constraints attribute codes is given below, along with the range of valid values for each attribute, which of these values are considered least and most restrictive, and the default value for the attribute. This is presented in the form of a table for each attribute category; each table is followed by detailed descriptions of each attribute.

In addition to the individual attributes specified for DMA operations there are a set of convenience attributes, which are used to specify a grouping of individual attributes. For most DMA situations, there are sets of related attributes which can all be set to identical values because the DMA engine does not distinguish between the conditions represented by the individual attributes. UDI provides convenience attributes that refer to a group of individual attributes and when used, causes all the individual attributes which are part of the group to be set at once without requiring explicit specification of each individual attribute.

Table 3-1 DMA Convenience Attributes

Convenience Attribute	**Related Individual Attributes**
UDI_DMA_ADDRESSABLE_BITS	UDI_DMA_DATA_ADDRESSABLE_BITS UDI_DMA_SCGTH_ADDRESSABLE_BITS
UDI_DMA_ALIGNMENT_BITS	UDI_DMA_ELEMENT_ALIGNMENT_BITS UDI_DMA_SCGTH_ALIGNMENT_BITS

UDI_DMA_ADDRESSABLE_BITS is used to simultaneously set UDI_DMA_DATA_ADDRESSABLE_BITS and UDI_DMA_SCGTH_ADDRESSABLE_BITS. This specifies the number of bits of bus address that the DMA engine can generate for access to either data or scatter/gather elements.

UDI_DMA_ALIGNMENT_BITS is used to simultaneously set UDI_DMA_ELEMENT_ALIGNMENT_BITS and UDI_DMA_SCGTH_ALIGNMENT_BITS. This specifies the # of LSB bits that must be zero in the starting bus addresses of data elements and scatter/gather segments; i.e. the starting bus addresses of data elements and scatter/gather segments must be multiples of 2^UDI_DMA_ALIGNMENT_BITS.

Table 3-2 DMA Constraints on the Entire Transfer

Valid Range	**Least Restrictive Value**	**Most Restrictive Value**	**Default Value**	**Special Case Behavior for 0**
	`UDI_DMA_DATA_ADDRESSABLE_BITS`			
16..255	255	16	255	N/A
	`UDI_DMA_NO_PARTIAL`			
0..1	0	1	0	N/A

UDI_DMA_DATA_ADDRESSABLE_BITS is the # of bits of bus address that the DMA engine can generate for access to data elements.

UDI_DMA_NO_PARTIAL is a flag indicating (if non-zero) that the device and/or driver cannot handle partial DMA mappings from `udi_dma_buf_map`. Either the entire request must be mapped in one call, or it must be failed.

Table 3-3 DMA Constraints on the Whole Scatter/Gather List

Valid Range	Least Restrictive Value	Most Restrictive Value	Default Value	Special Case Behavior for 0
		UDI_DMA_SCGTH_MAX_ELEMENTS		
0..65535	0	1	0	no restriction
		UDI_DMA_SCGTH_FORMAT		
see below	N/A	N/A	see below	N/A
		UDI_DMA_SCGTH_ENDIANNESS		
see below	N/A	N/A	see below	N/A
		UDI_DMA_SCGTH_ADDRESSABLE_BITS		
16..255	255	16	255	N/A
		UDI_DMA_SCGTH_MAX_SEGMENTS		
0..255	0	1	0	no restriction

UDI_DMA_SCGTH_MAX_ELEMENTS is the maximum # of elements that can be handled in one scatter/gather list. For DMA engines without scatter/gather support, this should be set to 1.

UDI_DMA_SCGTH_FORMAT determines the format of the scatter/gather list. It must be set to one of the legal values for the ***scgth_format*** member of udi_scgth_t, described on page 3-11. Both `UDI_SCGTH_32` and `UDI_SCGTH_64` may be set if the device supports both; each returned scatter/gather list's ***scgth_format*** member will indicate which mapping was used for that scatter/gather list (only one is used for the entire list and a 64-bit mapping is preferred).

The default value for this attribute is UDI_SCGTH_DMA_MAPPED + UDI_SCGTH_32.

The following attributes are relevant only to DMA-mapped scatter/gather lists and need not be set if UDI_DMA_SCGTH_FORMAT does not include UDI_SCGTH_DMA_MAPPED:

UDI_DMA_SCGTH_ENDIANNESS
UDI_DMA_SCGTH_ADDRESSABLE_BITS
UDI_DMA_SCGTH_ALIGNMENT_BITS
UDI_DMA_SCGTH_MAX_SEGMENTS
UDI_DMA_SCGTH_MAX_EL_PER_SEG
UDI_DMA_SCGTH_PREFIX_BYTES

UDI_DMA_SCGTH_ENDIANNESS determines the desired device endianness of DMA-mapped scatter/gather elements. This attribute may be set to one of the following values:

`UDI_DMA_LITTLE_ENDIAN`
`UDI_DMA_BIG_ENDIAN`

The default value for this attribute is unspecified. If UDI_DMA_SCGTH_FORMAT includes UDI_SCGTH_DMA_MAPPED, this attribute must be explicitly set before the constraints object can be used with `udi_dma_prepare` or `udi_dma_mem_alloc`.

Ignored if UDI_DMA_SCGTH_FORMAT does not include UDI_SCGTH_DMA_MAPPED.

UDI_DMA_SCGTH_ADDRESSABLE_BITS is the # of bits of bus address that the DMA engine can generate for accesses to scatter/gather list elements.

Ignored if UDI_DMA_SCGTH_FORMAT does not include UDI_SCGTH_DMA_MAPPED.

UDI_DMA_SCGTH_MAX_SEGMENTS is the maximum # of scatter/gather segments that the DMA engine can handle. (One segment references another through indirect scatter/gather elements.)

Ignored if UDI_DMA_SCGTH_FORMAT does not include UDI_SCGTH_DMA_MAPPED.

Table 3-4 DMA Constraints on Scatter/Gather Segments

Valid Range	Least Restrictive Value	Most Restrictive Value	Default Value	Special Case Behavior for 0
	`UDI_DMA_SCGTH_ALIGNMENT_BITS`			
0..255	0	255	0	N/A
	`UDI_DMA_SCGTH_MAX_EL_PER_SEG`			
0..65535	0	1	0	no restriction
	`UDI_DMA_SCGTH_PREFIX_BYTES`			
0..65535	0	65535	0	N/A

UDI_DMA_SCGTH_ALIGNMENT_BITS is the # of LSB bits that must be zero in the starting bus address of each scatter/gather segment); i.e. the starting bus address of each segment must be a multiple of 2^UDI_DMA_SCGTH_ALIGNMENT_BITS.

Ignored if UDI_DMA_SCGTH_FORMAT does not include UDI_SCGTH_DMA_MAPPED.

UDI_DMA_SCGTH_MAX_EL_PER_SEG is the maximum # of scatter/gather elements that can be handled in one segment, not including chain pointers to the next segment if any.

Ignored if UDI_DMA_SCGTH_FORMAT does not include UDI_SCGTH_DMA_MAPPED.

UDI_DMA_SCGTH_PREFIX_BYTES is the # bytes of extra uninitialized DMA-mapped memory that will be allocated preceding the actual scatter/gather element array in each scatter/gather segment. This is typically used to provide additional shared control structures for communicating with the device, or as space for conversion to alternate (larger) scatter/gather formats.

Alignment constraints on the starting address of scatter/gather segments will be applied to the beginning of this prefix area rather than the beginning of the element array.

From the DMA device, the prefix bytes will be accessible at bus addresses immediately preceding the bus address of the start of the segment.

Driver access to the memory, if needed, is via the same mechanism used for access to the scatter/gather elements, through the ***`scgth_elements`*** pointers. In this case, the scatter/gather list must also be driver-mapped.

Ignored if UDI_DMA_SCGTH_FORMAT does not include UDI_SCGTH_DMA_MAPPED.

Table 3-5 DMA Constraints on Individual Scatter/Gather Elements

Valid Range	Least Restrictive Value	Most Restrictive Value	Default Value	Special Case Behavior for 0
		`UDI_DMA_ELEMENT_ALIGNMENT_BITS`		
0..255	0	255	0	no restriction
		`UDI_DMA_ELEMENT_LENGTH_BITS`		
0..32	0	1	0	no restriction
		`UDI_DMA_ELEMENT_GRANULARITY_BITS`		
0..32	0	1	0	no restriction

UDI_DMA_ELEMENT_ALIGNMENT_BITS is the # of LSB bits that must be zero in the starting bus address of each individual scatter/gather element; i.e. the starting bus address must be a multiple of 2^UDI_DMA_ELEMENT_ALIGNMENT_BITS.

UDI_DMA_ELEMENT_LENGTH_BITS is the # bits supported for the length (in bytes) of a single scatter/gather element (***`block_length`*** in `udi_scgth_t`). No matter what this parameter is set to, no more than UDI_DMA_ADDRESSABLE_BITS will be used.

UDI_DMA_ELEMENT_GRANULARITY_BITS is the granularity of each individual scatter/gather element. The length in bytes of each element, except the last one in the whole transfer, must be a multiple of 2^UDI_DMA_ELEMENT_GRANULARITY_BITS.

Table 3-6 DMA Constraints for Special Addressing Restrictions

Valid Range	Least Restrictive Value	Most Restrictive Value	Default Value	Special Case Behavior for 0
		`UDI_DMA_ADDR_FIXED_BITS`		
0..255	0	1	0	no fixed bits
		`UDI_DMA_ADDR_FIXED_TYPE`		
see below	see below	see below	see below	N/A
		`UDI_DMA_ADDR_FIXED_VALUE_LO`		
0..2^{32}-1	N/A	N/A	0	see below
		`UDI_DMA_ADDR_FIXED_VALUE_HI`		
0..2^{32}-1	N/A	N/A	0	see below

UDI_DMA_ADDR_FIXED_BITS if non-zero, indicates that some number of MSB bits are "fixed"; that is, they must have the same value for all addresses in a range. Depending on the value of UDI_DMA_ADDR_FIXED_TYPE the specified bits may be "fixed" across an entire scatter/gather list or just within each element. In most cases, it doesn't matter what the "fixed" value is, as long as it is unchanging across the required range.

The value of UDI_DMA_ADDR_FIXED_BITS indicates the number of "variable" bits less significant than the least significant "fixed" bit. (This can also be thought of as the bit number of the first fixed bit, counting from the LSB, starting with 0.) All bits more significant than the first "fixed" bit, up to the most significant bit according to UDI_DMA_ADDRESSABLE_BITS, are also "fixed".

This somewhat counter-intuitive encoding is unfortunately necessary to prevent the value from being invalidated by changes to UDI_DMA_ADDRESSABLE_BITS.

The following attributes are relevant only when UDI_DMA_ADDR_FIXED_BITS is non-zero and need not be set otherwise:

UDI_DMA_ADDR_FIXED_TYPE
UDI_DMA_ADDR_FIXED_VALUE_LO
UDI_DMA_ADDR_FIXED_VALUE_HI

UDI_DMA_ADDR_FIXED_TYPE specifies the type of "fixed" address restriction, if any. It takes one of the following values:

UDI_DMA_FIXED_ELEMENT:
Each scatter/gather element may have different "fixed" values. This would typically be for a case where the DMA engine's current address register has upper bits that don't cascade, such as ISA motherboard DMAC "page" registers.

UDI_DMA_FIXED_LIST:
The whole scatter/gather list must have the same "fixed" value. Seems an unusual case, but it's here for completeness.

UDI_DMA_FIXED_VALUE:
The "fixed" bits must be equal to the value given by the pair of UDI_DMA_ADDR_FIXED_VALUE_LO and UDI_DMA_ADDR_FIXED_VALUE_HI for the whole scatter/gather list. This would only be used by special bridge drivers for some case like multiple OS instances sharing the same bus, in which the bridge driver "knows better" than the OS since it knows it was configured to allow only a sub-portion of the bus address space to be used by this instance.

UDI_DMA_FIXED_ELEMENT is least restrictive and the default. UDI_DMA_FIXED_VALUE is the most restrictive.

Ignored if UDI_DMA_ADDR_FIXED_BITS is zero.

UDI_DMA_ADDR_FIXED_VALUE_LO is the least significant 32 bits of the required "fixed" bits value in the UDI_DMA_FIXED_VALUE case; the LSB of UDI_DMA_ADDR_FIXED_VALUE_LO corresponds to the first "fixed" bit, not to the LSB of the address as a whole.

Ignored if UDI_DMA_ADDR_FIXED_BITS is zero or UDI_DMA_ADDR_FIXED_TYPE is not UDI_DMA_FIXED_VALUE.

UDI_DMA_ADDR_FIXED_VALUE_HI is the most significant 32 bits of the required "fixed" bits value in the UDI_DMA_FIXED_VALUE case. This should rarely ever need to be set.

Ignored if UDI_DMA_ADDR_FIXED_BITS is zero or UDI_DMA_ADDR_FIXED_TYPE is not UDI_DMA_FIXED_VALUE or UDI_DMA_ADDRESSABLE_BITS minus UDI_DMA_FIXED_BITS is not greater than 32.

Table 3-7 DMA Constraints on DMA Access Behavior

Valid Range	Least Restrictive Value	Most Restrictive Value	Default Value	Special Case Behavior for 0
		`UDI_DMA_SEQUENTIAL`		
0..1	0	1	0	N/A
		`UDI_DMA_SLOP_IN_BITS`		
0..8	0	8	0	N/A
		`UDI_DMA_SLOP_OUT_BITS`		
0..8	0	8	0	N/A
		`UDI_DMA_SLOP_OUT_EXTRA`		
0..65535	0	65535	0	N/A
		`UDI_DMA_SLOP_BARRIER_BITS`		
0..255	1	0	1	no slop barrier

UDI_DMA_SEQUENTIAL is a flag indicating (if non-zero) that the DMA engine accesses memory sequentially for a data transfer, in the order listed in the scatter/gather list and in sequential address order within each element. Some environment implementations can optimize DMA transfers if they get this guarantee from the driver.

UDI_DMA_SLOP_IN_BITS is the worst-case # bits of "input slop" alignment. This indicates that DMA cycles may cause accesses to additional physical memory locations besides those explicitly listed in scatter/gather elements. Specifically, the starting (inclusive) and ending (exclusive) byte addresses of each scatter/gather element may be rounded out to boundaries that are multiples of 2^UDI_DMA_SLOP_IN_BITS for "inbound" bus transactions (i.e. from device to memory).

UDI_DMA_SLOP_OUT_BITS is the worst-case # bits of "output slop" alignment. This indicates that DMA cycles may cause accesses to additional physical memory locations besides those explicitly listed in scatter/gather elements. Specifically, the starting (inclusive) and ending (exclusive) byte addresses of each scatter/gather element may be rounded out to boundaries that are multiples of 2^UDI_DMA_SLOP_OUT_BITS for "outbound" bus transactions (i.e. from memory to device).

UDI_DMA_SLOP_OUT_EXTRA is the worst-case # bytes of "extra slop". This represents additional bytes of prefetch beyond the end of each scatter/gather element after rounding up based on UDI_DMA_SLOP_OUT_BITS. This applies during outbound bus transactions only.

UDI_DMA_SLOP_BARRIER_BITS is the # bits of "slop barrier". This represents a boundary across which UDI_DMA_SLOP_OUT_EXTRA prefetching will not occur; i.e. prefetching will not cross address boundaries that are multiples of 2^UDI_DMA_SLOP_BARRIER_BITS. This is useful to alleviate the worst-case assumptions of UDI_DMA_SLOP_OUT_EXTRA.

Ignored if UDI_DMA_SLOP_OUT_EXTRA is zero.

REFERENCES

`udi_dma_constraints_attr_t`, `udi_dma_constraints_t`, `udi_buf_t`, `udi_dma_mem_alloc`, `udi_dma_prepare`, `udi_dma_constraints_attr_set`

NAME

udi_dma_prepare *Prepare for DMA mapping*

SYNOPSIS

```
#include <udi.h>
#include <udi_physio.h>

void udi_dma_prepare (
     udi_dma_prepare_call_t *callback,
     udi_cb_t *gcb,
     udi_dma_constraints_t constraints,
     udi_ubit8_t flags );

typedef void udi_dma_prepare_call_t (
     udi_cb_t *gcb,
     udi_dma_handle_t new_dma_handle );

/* Values for flags */
#define UDI_DMA_OUT                         (1U<<2)
#define UDI_DMA_IN                          (1U<<3)
```

ARGUMENTS

callback, gcb are standard arguments described in the “Asynchronous Service Calls” section of *“Standard Calling Sequences”* in the UDI Core Specification.

constraints is a constraints handle for this device.

flags is a bitmask of flags indicating the direction(s) of transfers for which the handle will most likely be used:

UDI_DMA_OUT	data transfer from memory to device
UDI_DMA_IN	data transfer from device to memory

new_dma_handle is an opaque handle to the newly allocated DMA object.

DESCRIPTION

`udi_dma_prepare` allocates a DMA handle that can be used to map UDI buffers for DMA transfer. In some cases DMA resources such as mapping registers will be pre-allocated at this time. The new DMA handle is passed to the driver with the callback.

It is intended that drivers avoid using `udi_dma_prepare` in the main I/O path. Where possible, it should be used at bind time, with many calls to `udi_dma_buf_map` being made for one call to `udi_dma_prepare`.

REFERENCES

`udi_dma_buf_map, udi_dma_free, udi_dma_constraints_t`

NAME

udi_dma_buf_map *Map a buffer for DMA*

SYNOPSIS

```
#include <udi.h>
#include <udi_physio.h>

void udi_dma_buf_map (
     udi_dma_buf_map_call_t *callback,
     udi_cb_t *gcb,
     udi_dma_handle_t dma_handle,
     udi_buf_t *buf,
     udi_size_t offset,
     udi_size_t len,
     udi_ubit8_t flags );

typedef void udi_dma_buf_map_call_t (
     udi_cb_t *gcb,
     udi_scgth_t *scgth,
     udi_boolean_t complete,
     udi_status_t status );

/* Values for flags */
#define UDI_DMA_OUT                      (1U<<2)
#define UDI_DMA_IN                       (1U<<3)
#define UDI_DMA_REWIND                   (1U<<4)
```

ARGUMENTS

callback, gcb are standard arguments described in the "Asynchronous Service Calls" section of *"Standard Calling Sequences"* in the UDI Core Specification.

dma_handle is the UDI DMA handle.

buf is a pointer to the UDI buffer containing the data request.

offset is the offset, in bytes, from the first valid data byte in ***buf*** at which to begin the DMA mapping. This must be less than or equal to `buf->buf_size`.

len is the number of bytes to map for DMA. If ***flags*** includes UDI_DMA_IN and there are fewer than ***len*** bytes of valid data in the buffer, starting at ***offset***, then the buffer will be extended but the initial values of the new data bytes are unspecified. If ***flags*** does not include UDI_DMA_IN, ***offset*** plus ***len*** must be less than or equal to the valid data length of the buffer.

flags is a bitmask of one or more flags indicating the direction(s) and range of transfers for which the mapping applies:

UDI_DMA_OUT	data transfer from memory to device
UDI_DMA_IN	data transfer from device to memory
UDI_DMA_REWIND	rewind to the beginning of the buffer

scgth is a pointer to a DMA scatter/gather structure.

complete is a boolean flag indicating whether the buffer was completely or partially mapped for DMA.

status is a UDI status code indicating the success or failure of the service call. If not UDI_OK, the ***scgth*** and ***complete*** values are unspecified and must be ignored.

DESCRIPTION

`udi_dma_buf_map` allocates any additional DMA resources required to prepare the specified buffer for a DMA transfer and passes back a pointer to a DMA scatter/gather structure. The `udi_scgth_t` contains a set of bus address/length pairs, referencing the DMA target, which the driver may use as is, translate in place, or copy to the scatter/gather entries of its device's DMA engine, as is appropriate for its device. This scatter/gather list covers the ***len*** mapped bytes of the buffer, starting at ***offset***.

The memory referenced by the ***scgth*** is guaranteed to meet the DMA constraints indicated by the ***constraints*** argument passed to `udi_dma_prepare` when ***dma_handle*** was allocated; these constraints take precedence over those associated with the buffer. Any data modified through the DMA mapping will be visible through the buffer handle once it is unmapped with `udi_dma_buf_unmap`.

While DMA-mapped, the buffer data must not be accessed via the buffer handle. The driver may change the buffer's `buf_size` value, however, but it will be ignored and ultimately overwritten by the value specified for the `udi_dma_buf_unmap` call.

The first time `udi_dma_buf_map` is called for a particular buffer using this DMA handle, the data is mapped starting at the beginning of the buffer. If the full size cannot be mapped in one piece, the ***complete*** flag is set to FALSE, indicating a partial transfer. Once this piece is complete, the driver can call `udi_dma_buf_map` again with the same arguments. The environment will "remember" where it left off (typically using information in the DMA handle) and will map the next section of the buffer.

To restart at the beginning of the buffer after a partial transfer, set the `UDI_DMA_REWIND` flag. The first call to `udi_dma_buf_map` for a particular buffer always starts at the beginning of the buffer, whether or not the `UDI_DMA_REWIND` flag is set.

At least one of `UDI_DMA_IN` or `UDI_DMA_OUT` must be specified in the ***flags*** argument, optionally combined with `UDI_DMA_REWIND`.

`udi_dma_buf_map` performs an implicit `udi_dma_sync` for the entire mapped range with the same `UDI_DMA_IN` and/or `UDI_DMA_OUT` ***flags*** as passed to `udi_dma_buf_map`.

STATUS VALUES

`UDI_OK` is returned to indicate that the mapping completed successfully.

`UDI_STAT_RESOURCE_UNAVAIL` is returned to indicate a partial mapping. This value is only returned if `UDI_DMA_NO_PARTIAL` was not set in the constraints that were used to allocate the DMA handle and the environment is unable to map the entire request at one time. This status code indicates that the mapping has failed, the ***`scgth`*** pointer is NULL, and ***`dma_handle`*** is still unused, although ***`new_buf`*** may still be different than the original ***`buf`*** pointer.

REFERENCES `udi_dma_buf_unmap, udi_dma_prepare, udi_scgth_t`

NAME

udi_dma_buf_unmap *Release a buffer's DMA mapping*

SYNOPSIS

```
#include <udi.h>
#include <udi_physio.h>

udi_buf_t *udi_dma_buf_unmap (
     udi_dma_handle_t dma_handle,
     udi_size_t new_buf_size );
```

ARGUMENTS

dma_handle is a DMA handle previously mapped via `udi_dma_buf_map`.

new_buf_size is the number of bytes of data to preserve from the mapped buffer. This becomes the new value of `buf->buf_size` for the buffer.

DESCRIPTION

`udi_dma_buf_unmap` frees any resources associated with a DMA handle by a previous `udi_dma_buf_map` request. It should be used when a DMA transfer completes and its DMA handle is not going to be reused with the associated buffer.

If the ***flags*** passed to `udi_dma_buf_map` for this handle included `UDI_DMA_IN`, `udi_dma_buf_unmap` performs an implicit inbound `udi_dma_sync` for the entire mapped range and ensures that any data modified by the device is now visible to the driver.

If ***dma_handle*** is equal to `UDI_NULL_DMA_HANDLE`, explicitly or implicitly (zeroed by initial value or by using `udi_memset`), this function acts as a no-op. Otherwise, ***dma_handle*** must have been allocated by `udi_dma_prepare`.

Even if the buffer's ***buf_size*** value was changed while the buffer was mapped, the entire buffer will be unmapped. Any buffer data beyond ***new_buf_size*** at the time of the `udi_dma_buf_unmap` will be discarded (though the memory might not be).

WARNINGS

The driver must make sure that its device is no longer accessing the buffer or control structure memory before it calls `udi_dma_buf_unmap`.

RETURN VALUES

The `udi_dma_buf_unmap` function returns a buffer pointer that the driver must now use in place of the original ***buf.*** This is logically the same buffer that was passed to `udi_dma_buf_map`, but the environment may have reallocated it in the process of handling the DMA operation.

REFERENCES

`udi_dma_buf_map, udi_dma_prepare`

NAME

udi_dma_mem_alloc *Allocate shared control structure memory*

SYNOPSIS

```
#include <udi.h>
#include <udi_physio.h>

void udi_dma_mem_alloc (
     udi_dma_mem_alloc_call_t *callback,
     udi_cb_t *gcb,
     udi_dma_constraints_t constraints,
     udi_ubit8_t flags,
     udi_ubit16_t nelements,
     udi_size_t element_size,
     udi_size_t max_gap );

typedef void udi_dma_mem_alloc_call_t (
     udi_cb_t *gcb,
     udi_dma_handle_t new_dma_handle,
     void *mem_ptr,
     udi_size_t actual_gap,
     udi_boolean_t single_element,
     udi_scgth_t *scgth,
     udi_boolean_t must_swap );

/* Values for flags */
#define UDI_DMA_OUT                       (1U<<2)
#define UDI_DMA_IN                        (1U<<3)
#define UDI_DMA_BIG_ENDIAN                (1U<<5)
#define UDI_DMA_LITTLE_ENDIAN             (1U<<6)
#define UDI_DMA_NEVERSWAP                 (1U<<7)
```

ARGUMENTS

callback, gcb are standard arguments described in the "Asynchronous Service Calls" section of *"Standard Calling Sequences"* in the UDI Core Specification.

constraints is a UDI constraints handle that defines the memory constraints of the DMA engine that will be accessing the allocated memory.

flags is a bitmask of flags, that control the operation of this function.

These flags must include one or both of the following direction flags, which indicate the direction(s) of DMA transfers that may be used:

UDI_DMA_IN - transfers from device to memory.

UDI_DMA_OUT - transfers from memory to device.

The flags must also include exactly one of:

UDI_DMA_BIG_ENDIAN - access data in big endian format.

UDI_DMA_LITTLE_ENDIAN - access data in little endian format.

UDI_DMA_NEVERSWAP - access data with no byte swapping (appropriate for character data).

Finally, ***flags*** may optionally include UDI_MEM_NOZERO.

nelements indicates the number of memory elements that are to be allocated.

element_size indicates the size of each element to be allocated.

max_gap indicates the maximum number of bytes that may separate each element from the last byte of the preceeding element to the first byte of the next element. A driver will typically set this to indicate how far apart the elements may be for the device to properly address them; a ***max_gap*** of zero indicates that the elements must be adjacent in memory.

new_dma_handle is an opaque handle to the newly allocated DMA object.

mem_ptr is a driver-mapped pointer to the allocated DMA-able memory.

actual_gap indicates the actual number of bytes that separate each control structure in memory. This value will be equal to or less than the requested ***max_gap*** value unless the environment could not satisfy the requested parameters, in which case ***single_element*** will be TRUE and the ***actual_gap*** argument must be ignored. This argument must also be ignored if ***nelements*** is 1.

single_element is a flag indicating whether all elements were allocated or whether only a single element was allocated. This will return true when the allocation specified by the arguments, including the specified constraints, cannot be satisfied; in this case only one element is allocated and returned. This argument must be ignored if ***nelements*** is 1.

scgth is a pointer to a DMA scatter/gather structure for the newly allocated memory.

must_swap is a flag indicating that the driver must swap endianness when it accesses the DMA-able memory via ***mem_ptr***. This will be computed by the environment as a function of the driver's endianness, the device endianness specified in ***flags***, and any interceding bus bridges. Because bridges may introduce additional endianness changes, drivers must always check this flag rather than assuming swapping or not swapping.

See Section 22.2, "Endianness Management," on page 22-2 of the UDI Core Specification for details on how to construct C structure definitions for proper endianness handling, and the endian swapping utilities that are available.

DESCRIPTION

Allocates control structure memory that is to be shared between a UDI device driver and its corresponding DMA device. This memory will conform to the requirements expressed in ***constraints***. The device accesses the memory via DMA. The driver accesses the memory using the ***mem_ptr*** pointer, and must handle endian conversions if so indicated by the ***must_swap*** flag.

If the driver will be using the memory for multiple separate control structures, the ***nelement*** and ***element_size*** arguments should be used to describe the individual control structures. The system will adjust the allocation to ensure that each control structure: (1) starts on an appropriate alignment boundary, (2) does not share cache lines with other control structures, and (3) is physically contiguous. This adjustment is indicated by the ***actual_gap*** value returned on the callback which indicates the number of additional bytes allocated between each control structure to satisfy the individual alignment requirements. This gap will not exceed the device's capabilities as indicated by the ***max_gap*** argument; if the required gap size would exceed this value, then only a single element is allocated as indicated by the ***single_element*** argument.

Each gap represents actual allocated memory bytes; these bytes appear in both the virtual and the DMA-mapped address ranges and the driver should make appropriate considerations for accessing each element. There is no gap following the last element allocated.

The newly allocated memory will be zeroed unless UDI_MEM_NOZERO is set, in which case the initial values are unspecified.

The newly allocated memory will be aligned on the most restrictive alignment of the platform's natural alignments for *long* and pointer data types, allowing the allocated memory to be directly accessed as C structures.

At the time of the callback, the memory is mapped for DMA access by the device. The DMA mapped memory remains allocated and mapped until ***new_dma_handle*** is freed by a call to `udi_dma_free` or `udi_dma_mem_to_buf`, at which point the memory will be automatically unmapped and deallocated. `udi_dma_mem_to_buf` allows a range of data from the control structure memory to be placed into a UDI buffer.

Note – Unlike `udi_dma_buf_map`, `udi_dma_mem_alloc` will always produce a complete mapping.

This call is typically used for allocating memory that is contiguous from the device's perspective, but some devices may support discontiguous control memory. Whether or not contiguous device addresses are used is under control of the UDI_DMA_SCGTH_MAX_ELEMENTS property of the constraints handle, but the memory will always be virtually contiguous when accessed through the ***mem_ptr*** pointer. See `udi_dma_constraints_attr_t` (DMA) for details on DMA constraints.

Note – In order to use the byte-by-byte structure layout technique for mixed endianness access to shared control structure memory, and the utility macros defined in Section 22.2, "Endianness Management," on page 22-2 of the UDI Core Specification, the driver must declare two versions of the relevant structure(s), type-casting appropriately, and using the version that matches the device's endianness if ***`must_swap`*** is FALSE, or the "anti-endian" version if ***`must_swap`*** is TRUE.

REFERENCES `udi_dma_limits`, `udi_dma_constraints_attr_t`, `udi_dma_prepare`, `udi_dma_buf_map`, `udi_dma_free`, `udi_dma_sync`, `udi_dma_mem_barrier`, `udi_scgth_t`, `udi_dma_mem_to_buf`

NAME

udi_dma_sync *Sync host & device views of DMA-able memory*

SYNOPSIS

```
#include <udi.h>
#include <udi_physio.h>

void udi_dma_sync (
     udi_dma_sync_call_t *callback,
     udi_cb_t *gcb,
     udi_dma_handle_t dma_handle,
     udi_size_t offset,
     udi_size_t length,
     udi_ubit8_t flags );

typedef void udi_dma_sync_call_t (
     udi_cb_t *gcb );

/* Values for flags */
#define UDI_DMA_OUT                       (1U<<2)
#define UDI_DMA_IN                        (1U<<3)
```

ARGUMENTS

callback, gcb are standard arguments described in the "Asynchronous Service Calls" section of *"Standard Calling Sequences"* in the UDI Core Specification.

dma_handle is a DMA handle previously mapped via `udi_dma_buf_map` or `udi_dma_mem_alloc`.

offset is a logical offset into the mapped buffer data. That is, it is relative to the ***offset*** value provided to `udi_dma_buf_map`.

length is the length, in bytes, of the area to synchronize. When length is zero, it applies to the entire data object referenced by ***dma_handle***, and ***offset*** must be zero.

flags indicates which view of the buffer to synchronize. The ***flags*** argument may be set to one or more of the following values:

UDI_DMA_OUT - sync before starting an outbound DMA to the device

UDI_DMA_IN - sync after completing an inbound DMA from the device

DESCRIPTION

`udi_dma_sync` is used to synchronize the host and device views of a data object that has been loaded for DMA. This may involve flushes of CPU or I/O caches, or assuring that hardware write buffers have drained.

The required direction flags depend on the direction(s) of I/O transactions since the last synchronization points. If the buffer has been modified by the CPU, and is going to be read by the device's DMA engine, then `udi_dma_sync` must be called with UDI_DMA_OUT set. This ensures that the device's DMA engine sees the changes previously made to the buffer memory by the driver. If the device's DMA engine has written to the buffer, and it is going to be read by the driver, `udi_dma_sync` must be called with UDI_DMA_IN set. This makes sure the CPU's view of the memory includes any changes previously made by the device's DMA engine.

Note that the UDI_DMA_OUT and/or UDI_DMA_IN flags must have been set in the call to `udi_dma_prepare` or `udi_dma_mem_alloc` if the same flag is to be set in `udi_dma_sync`.

REFERENCES

`udi_dma_buf_map`, `udi_dma_buf_unmap`, `udi_dma_mem_alloc`, `udi_dma_mem_barrier`, `udi_dma_scgth_sync`

NAME

udi_dma_scgth_sync *Sync host & device views of scatter/gather list*

SYNOPSIS

```
#include <udi.h>
#include <udi_physio.h>

void udi_dma_scgth_sync (
     udi_dma_scgth_sync_call_t *callback,
     udi_cb_t *gcb,
     udi_dma_handle_t  dma_handle );

typedef void udi_dma_scgth_sync_call_t (
     udi_cb_t *gcb );
```

ARGUMENTS

callback, gcb are standard arguments described in the "Asynchronous Service Calls" section of *"Standard Calling Sequences"* in the UDI Core Specification.

dma_handle is a DMA handle previously mapped via `udi_dma_buf_map` or `udi_dma_mem_alloc`.

DESCRIPTION

`udi_dma_scgth_sync` is used to synchronize the host and device views of the scatter/gather list memory for a data object that has been loaded for DMA, since `udi_dma_sync` only affects the actual data memory. This may involve flushes of CPU or I/O caches, or assuring that hardware write buffers have drained.

The entire set of scatter/gather elements, in all segments, as well as any prefix bytes (`UDI_DMA_SCGTH_PREFIX_BYTES` in **udi_dma_constraints_attr_t** on page 3-15) included in the synchronization. It is assumed that the DMA device did not write to this memory unless `UDI_DMA_SCGTH_PREFIX_BYTES` was greater than zero, but the driver may have read and/or written.

This function is only needed when the scatter/gather list is both DMA-mapped and driver-mapped (see **udi_scgth_t** on page 3-10), since this is the only case in which the driver will write to the scatter/gather segment memory and the device will read from it, and must not be used in other cases. In this case, `udi_dma_scgth_sync` must be called before the device reads from the scatter/gather list.

REFERENCES

`udi_dma_buf_map, udi_dma_mem_alloc, udi_dma_sync`

NAME

udi_dma_mem_barrier *Ordering barrier for accesses to DMA-able memory*

SYNOPSIS

```
#include <udi.h>
#include <udi_physio.h>

void udi_dma_mem_barrier (
     udi_dma_handle_t dma_handle );
```

ARGUMENTS

dma_handle is a DMA handle previously allocated by `udi_dma_mem_alloc`.

DESCRIPTION

`udi_dma_mem_barrier` is used to impose ordering constraints between driver accesses to shared control structure memory allocated by `udi_dma_mem_alloc`.

This ensures that no loads or stores to the memory associated with ***dma_handle*** that are executed after the call to `udi_dma_mem_barrier` will be visible to the device until all prior loads or stores are visible.

REFERENCES

`udi_dma_mem_alloc, udi_dma_sync`

NAME

udi_dma_free *Free DMA resources*

SYNOPSIS

```
#include <udi.h>
#include <udi_physio.h>

void udi_dma_free (
     udi_dma_handle_t dma_handle );
```

ARGUMENTS

dma_handle **is a DMA handle allocated by** `udi_dma_prepare` **or** `udi_dma_mem_alloc`**.**

DESCRIPTION

`udi_dma_free` frees a DMA handle and any associated resources allocated by `udi_dma_prepare` or `udi_dma_mem_alloc`.

If the ***dma_handle*** is associated with shared control structure memory allocated by `udi_dma_mem_alloc`, the allocated memory will be freed, as will the scatter/gather list.

If the ***dma_handle*** was mapped to a buffer with `udi_dma_buf_map`, it must be unmapped (using `udi_dma_buf_unmap`) before `udi_dma_free` is called.

If ***dma_handle*** is equal to `UDI_NULL_DMA_HANDLE`, explicitly or implicitly (zeroed by initial value or by using `udi_memset`), this function acts as a no-op. Otherwise, ***dma_handle*** must have been allocated by `udi_dma_prepare` or `udi_dma_mem_alloc`.

Note – `udi_dma_free` does not do a `udi_dma_sync` operation.

WARNINGS

The driver must make sure that its device is no longer accessing the buffer or control structure memory before it calls `udi_dma_free`.

REFERENCES

`udi_dma_prepare, udi_dma_buf_map, udi_dma_buf_unmap, udi_dma_mem_alloc`

NAME

udi_dma_mem_to_buf *Convert DMA-mapped control memory into a buffer*

SYNOPSIS

```
#include <udi.h>
#include <udi_physio.h>

void udi_dma_mem_to_buf (
     udi_dma_mem_to_buf_call_t *callback,
     udi_cb_t *gcb,
     udi_dma_handle_t dma_handle,
     udi_size_t src_off,
     udi_size_t src_len,
     udi_buf_t *dst_buf );

typedef void udi_dma_mem_to_buf_call_t (
     udi_cb_t *gcb,
     udi_buf_t *new_dst_buf );
```

ARGUMENTS

callback, gcb are standard arguments described in the “Asynchronous Service Calls” section of *“Standard Calling Sequences”* in the UDI Core Specification.

dma_handle is a DMA handle previously allocated by `udi_dma_mem_alloc`.

src_off is the beginning offset from the first byte of the shared control structure memory to include in the destination buffer.

src_len is the number of bytes to include from the destination buffer.

dst_buf is the same argument as used in `udi_buf_copy`.

new_dst_buf is a pointer to a UDI buffer containing the indicated data.

DESCRIPTION

`udi_dma_mem_to_buf` frees a DMA handle and any associated resources allocated by `udi_dma_mem_alloc`, except the data from the indicated range of the memory block are used as the initial contents of a newly-allocated buffer.

The constraints that were used to allocate ***dma_handle*** shall also be used to allocate ***new_dst_buf***.

`udi_dma_mem_to_buf` is typically used for inbound data from devices that store both shared control structures and data in the same piece of memory.

REFERENCES

`udi_dma_mem_alloc, udi_buf_write`

3.4 DMA Constraints Handle Transferability

Since DMA Constraints handles are transferable between regions, they need to be represented in layout specifiers for use by metalanguages and by drivers that specify inline layouts. This section defines an extension to the data layout specifier defined in the UDI Core Specification.

NAME

udi_layout_t (DMA) *Data layout specifier for DMA*

SYNOPSIS

```
#include <udi.h>
#include <udi_physio.h>

typedef udi_ubit8_t udi_layout_t;

/* DMA Constraints Handle Layout Element Type Code */
#define UDI_DL_DMA_CONSTRAINTS_T          201
```

DESCRIPTION

This page lists additional layout specifier codes that can be used with the `udi_layout_t` type, defined in the UDI Core Specification, to specify DMA-related data layouts.

A data layout specifier consists of an array of one or more `udi_layout_t` layout elements. Each element contains a type code indicating one of the UDI data types that can be passed into a channel operation, either as a field in the control block or as an additional parameter. Each successive element of the array represents successive offsets within the described structure, with padding automatically inserted for alignment purposes as if the specified data types had appeared in a C `struct` declaration.

A `UDI_DL_DMA_CONSTRAINTS_T` layout element represents a DMA Constraints handle, of type `udi_dma_constraints_t`, which may be `UDI_NULL_DMA_CONSTRAINTS`.

REFERENCES

`udi_layout_t`, `udi_dma_constraints_t`

Programmed I/O (PIO) 4

4.1 Overview

Programmed I/O (PIO) refers to data transfers initiated by a CPU under driver software control to access registers or memory on a device. This is contrasted with Direct Memory Access (DMA), which involves transfers initiated by a device to access system memory. On some hardware platforms PIO is handled via normal memory loads and stores ("memory-mapped I/O"); on others it requires special I/O instructions. UDI hides this difference from drivers.

In UDI, Programmed I/O (PIO) operations are performed through environment service calls coded as function calls rather than by direct memory references or I/O instructions in the drivers. This abstraction is necessary for driver portability across different machine architectures for (at least) the following reasons:

1. Different machines support different endian architectures and some provide hardware-assisted endianness swapping.
2. Some machine architectures may restrict direct access.
3. Some bus bridges are non-transparent and require software intervention (via a bus bridge driver) for each PIO access.

To reduce the overhead associated with using function calls, UDI allows multiple PIO transactions to be performed with a single function call.

PIO access properties are encapsulated with a PIO handle (`udi_pio_handle_t`). A PIO handle is an opaque data type containing the addressing, data translation and access constraint information required to access a device or memory address in a particular address space. For example, for PCI devices the address space could be memory space, I/O space or configuration space. Information specifying endianness, required atomicity, and data ordering constraints is also contained in the PIO handle.

Also associated with each PIO handle is a transaction list, that specifies the PIO operations to be invoked when that handle is passed to `udi_pio_trans`.

For each transaction that accesses the device, a PIO offset is specified, which indicates the offset into the space referenced by a PIO handle at which an I/O operation is to occur. The space covered by a particular PIO handle is contiguous with respect to the addresses seen by the device.

Synchronization among different PIO transaction lists is defined by the ***`serialization_domain`*** argument to the PIO mapping call. The execution of a PIO transaction list is serialized with respect to the execution of all other PIO transaction lists mapped to the same device instance and serialization domain; i.e., for a given device and serialization domain, at most one thread of execution will be active executing a corresponding transaction list and each such transaction list will execute to completion before another transaction list for this serialization domain begins execution.

Note, however, that PIO trans lists do not have any serialization guarantees with respect to region execution. A sequence of PIO transactions, encapsulated in a PIO transaction list, is passed to the environment via `udi_pio_trans` and may be performed outside the context of the caller's region. As a result, the PIO sequence may be executed in parallel to code executing in the region.

There are no ordering guarantees with respect to the processing of transactions lists for separate `udi_pio_trans` calls except that calls made from the same region for the same serialization domain will be processed in FIFO order. Additionally, the callbacks for these `udi_pio_trans` calls are also called in FIFO order for that serialization domain, although the callback processing is not necessarily consecutive with the processing of the transaction list.

Synchronization and ordering requirements for PIO operations, with respect to system operations such as cache or I/O buffer flushes, are device and driver specific. Drivers control when such synchronization and ordering operations are performed through the use of UDI_PIO_SYNC and UDI_PIO_BARRIER transactions. Additional data ordering and synchronization requirements may be associated with a PIO handle via the ***`pio_attributes`*** parameter to `udi_pio_map`.

4.2 PIO Handle Allocation and Initialization

The following functions are used to allocate and initialize handles to PIO-accessible memory. The `udi_pio_map` function allocates a handle for a range of device memory/registers. The `udi_pio_unmap` function deallocates the PIO handle and any associated resources when the driver no longer needs them. The `udi_pio_atomic_sizes` function returns a bitmask of transaction sizes that can be handled atomically.

Additionally, the driver must register a special PIO handle with the environment via the `udi_pio_abort_sequence` function, which the environment can use when “killing” a faulting region to stop the corresponding device from initiating further actions.

NAME

udi_pio_handle_t *PIO handle type*

SYNOPSIS

```
#include <udi.h>
#include <udi_physio.h>

typedef <HANDLE> udi_pio_handle_t;

/* Null handle value for udi_pio_handle_t */
#define UDI_NULL_PIO_HANDLE         <NULL_HANDLE>
```

DESCRIPTION

The PIO handle type, `udi_pio_handle_t`, holds an opaque handle that refers to an environment object which contains addressing, data translation and access information, as well as a PIO transaction list.

PIO handles are transferable between regions.

Drivers will often have multiple PIO handles for different address spaces on the same device (e.g. one handle for configuration space, another for I/O registers, etc.). Drivers might also have multiple PIO handles for the same address space if different translation requirements or constraints exist within that address space, or if different types of operations are to be used at different times.

REFERENCES

`udi_pio_trans_t`

NAME

udi_pio_map *Map device memory/registers for access*

SYNOPSIS

```
#include <udi.h>
#include <udi_physio.h>

void udi_pio_map (
     udi_pio_map_call_t *callback,
     udi_cb_t *gcb,
     udi_ubit32_t regset_idx,
     udi_ubit32_t base_offset,
     udi_ubit32_t length,
     udi_pio_trans_t *trans_list,
     udi_ubit16_t list_length,
     udi_ubit16_t pio_attributes,
     udi_ubit32_t pace,
     udi_index_t serialization_domain );

typedef void udi_pio_map_call_t (
     udi_cb_t *gcb,
     udi_pio_handle_t new_pio_handle );

/* Values for pio_attributes */
#define UDI_PIO_STRICTORDER              (1U<<0)
#define UDI_PIO_UNORDERED_OK             (1U<<1)
#define UDI_PIO_MERGING_OK               (1U<<2)
#define UDI_PIO_LOADCACHING_OK           (1U<<3)
#define UDI_PIO_STORECACHING_OK          (1U<<4)
#define UDI_PIO_BIG_ENDIAN               (1U<<5)
#define UDI_PIO_LITTLE_ENDIAN            (1U<<6)
#define UDI_PIO_NEVERSWAP                (1U<<7)
#define UDI_PIO_UNALIGNED                (1U<<8)
```

ARGUMENTS

callback, gcb are standard arguments described in the “Asynchronous Service Calls” section of *“Standard Calling Sequences”* in the UDI Core Specification.

regset_idx is the index to the specified register set within a given address space. The definition of ***regset_idx*** is bus and device dependent, and must be determined by the driver via the *bus_type* instance attribute (see Chapter 15, *“Instance Attribute Management”* in the UDI Core Specification) in conjunction with the corresponding UDI bus bindings (see “Section 3: Bus Bindings”). For example, if *bus_type* is “`pci`” then the settings would be based on the definitions in the *UDI PCI Bus Binding Specification*.

base_offset is the offset into the selected register address space at which this mapping is to start. This offset must be a multiple of all transaction sizes used to access device addresses through this handle, unless `UDI_PIO_UNALIGNED` is set in ***pio_attributes***.

length is the length to be mapped in bytes.

trans_list is a list of one or more PIO transactions to apply to this device when using `udi_pio_trans` with this PIO handle. The memory pointed to by ***trans_list*** must be in a module-global (and thus read-only) variable.

list_length is the number of elements in the ***trans_list*** list.

pio_attributes are attribute flags that specify the data translation and ordering requirements of accesses to the device memory.

Data Translation Flags

Attribute flags specifying the data translation requirements of the device are mutually exclusive. At most one must be used in a given handle allocation. The default is UDI_PIO_NEVERSWAP if no data translation flags are specified.

The following data translation flags are defined:

UDI_PIO_BIG_ENDIAN - access data in big endian format.

UDI_PIO_LITTLE_ENDIAN - access data in little endian format.

UDI_PIO_NEVERSWAP - access data with no byte swapping (appropriate for character data). Device transactions greater than one byte in size are illegal if this flag is set.

Data Ordering Flags

Attribute flags specifying the data ordering requirements for the device may be used in combination. With the exception of UDI_PIO_STRICTORDER these values are advisory, not mandatory. For example, data can be ordered without being merged or cached, even though a driver permits unordered, merged and cached operation. Strict ordering may be required for certain I/O operations but can be very costly on high performance computers. For unordered execution, merging or caching, the UDI_PIO_SYNC operation (see `udi_pio_trans`) can provide more specific and efficient synchronization.

The default is UDI_PIO_STRICTORDER if no ordering flags are specified. If ***pace*** is non-zero, ***pio_attributes*** must not include any data ordering flags other than UDI_PIO_STRICTORDER.

The following data ordering flags are defined:

UDI_PIO_STRICTORDER - the CPU must issue the references in order, as the programmer specified. Strict ordering is the default if no other data ordering flags are set. UDI_PIO_STRICTORDER must not be combined with any other data ordering flags.

UDI_PIO_UNORDERED_OK - the CPU may reorder both load and store references to the mapped area.

UDI_PIO_MERGING_OK - merging and batching: the CPU may merge individual stores to consecutive locations (for example, turn two consecutive byte stores into one halfword store), and it may batch individual loads (for example, turn two consecutive byte loads into one halfword load). If UDI_PIO_MERGING_OK is set, UDI_PIO_UNORDERED_OK is treated as if it were set, even if it isn't actually set.

UDI_PIO_LOADCACHING_OK - load caching: the CPU may cache the data it fetches and reuse it until another store occurs. The default is to fetch new data on every load. If UDI_PIO_LOADCACHING_OK is set, UDI_PIO_MERGING_OK is treated as if it were set, even if it isn't actually set.

UDI_PIO_STORECACHING_OK - store caching: the CPU may keep the data in the cache and push it to the device (perhaps with other data) at a later time. The default is to push data immediately to the device. If UDI_PIO_STORECACHING_OK is set, UDI_PIO_LOADCACHING_OK is treated as if it were set, even if it isn't actually set.

Data Alignment Flags

UDI_PIO_UNALIGNED - unaligned accesses allowed: if this flag is set, there are no restrictions on ***base_offset*** and on PIO offsets for individual transactions; otherwise, both of these must be multiples of all transaction sizes used in ***trans_list***. If this flag is set, none of the transactions using this handle are guaranteed to be atomic.

pace is the PIO pacing time, in microseconds, for this handle. The environment will guarantee that any device access that occurs using this PIO handle will be followed by at least ***pace*** microseconds before another access occurs to the same device register set (via any handle). If non-zero, the UDI_PIO_STRICTORDER attribute must be specified.

PIO transactions may be posted/cached for future completion, so the apparent execution time of PIO operation calls may vary from the driver's perspective, but there will be at least ***pace*** microseconds between them from the hardware's perspective[1].

serialization_domain is a numeric value indicating the serialization domain for this PIO handle. Transaction lists for a given device with the same serialization domain are serialized with respect to each other, and will further be executed in FIFO order with respect to `udi_pio_trans` calls from a given region. See the ordering definitions on page 4-1 for additional details.

new_pio_handle is an opaque data type containing the addressing, endian translation, and access constraint information required to access the associated device memory.

DESCRIPTION

`udi_pio_map` initializes a handle through which a driver instance may access its device using PIO access routines. The ***regset_idx***, ***base_offset***, and ***length*** arguments select the range of device memory to be made accessible; the ***pio_attributes*** argument specifies the ways in which it may be accessed.

Note that the driver is allowed to map the same piece of device memory multiple times for different access requirements. For example, if the driver wants to do both strongly-ordered and weakly ordered accesses to a given register set it can map the register set once with UDI_PIO_STRICTORDER (strongly ordered accesses) and once with UDI_PIO_UNORDERED_OK (weakly ordered accesses). The driver would then use the "strong" PIO handle when it wants to do a strongly ordered access, and the "weak" handle otherwise. Similarly, multiple handles to a given register set could be used for zero pacing vs. various nonzero pacing values, and for other variations in attributes, offset, length, or trans list.

REFERENCES

`udi_pio_handle_t, udi_pio_unmap, udi_pio_trans`

1.The PIO pacing delay may be implemented in hardware or software by the environment, but represents a potential inline delay in region execution. The driver writer is strongly advised to use only the delays required for their device, as excessive use can have adverse effects on the driver and the rest of the system.

NAME

udi_pio_unmap *Unmap a PIO handle and free associated resources*

SYNOPSIS

```
#include <udi.h>
#include <udi_physio.h>

void udi_pio_unmap (
     udi_pio_handle_t pio_handle );
```

ARGUMENTS

pio_handle is the PIO handle to be freed.

DESCRIPTION

This service call unmaps a PIO access handle, freeing any associated resources. Upon return ***pio_handle*** is no longer valid.

If ***pio_handle*** is equal to `UDI_NULL_PIO_HANDLE`, either explicitly or implicitly (zeroed by initial value or by using `udi_memset`), this function acts as a no-op. Otherwise, ***pio_handle*** must have been allocated by `udi_pio_map`.

REFERENCES

`udi_pio_handle_t, udi_pio_map`

NAME

udi_pio_atomic_sizes *Retrieve supported PIO operation atomicity*

SYNOPSIS

```
#include <udi.h>
#include <udi_physio.h>

udi_ubit32_t udi_pio_atomic_sizes (
     udi_pio_handle_t pio_handle );
```

ARGUMENTS

pio_handle is a PIO handle previously acquired via `udi_pio_map`.

DESCRIPTION

This routine retrieves an encoding of the PIO atomic transaction sizes (see *atomic transaction* in the Glossary) for which atomicity is supported when doing PIO accesses through ***pio_handle***. The set of supported atomic sizes is platform and bus specific, and may also be dependent on device address space, and register set.

The driver will typically use this routine to verify up front that the atomic sizes it requires are supported, and then fail its initialization if they're not supported. If such a check is not done and the platform does not support the device's atomicity requirements then the results of corresponding PIO accesses will be indeterminate.

Atomicity is not guaranteed, regardless of transaction size, for any handle mapped with the `UDI_PIO_UNALIGNED` flag, or when using `udi_pio_probe`.

RETURN VALUES

A bit encoding of the PIO operation sizes for which atomicity is supported is returned. Each bit position corresponds to a power of two byte PIO operation size. If a bit is set PIO operations of the corresponding size are supported atomically through the bus hierarchy to the device. For example, a configuration supporting atomic PIO operations of 1, 2, and 4 bytes would return a binary 0111.

REFERENCES

`udi_pio_map`

NAME

udi_pio_abort_sequence *Register a PIO abort sequence*

SYNOPSIS

```
#include <udi.h>
#include <udi_physio.h>

void udi_pio_abort_sequence (
    udi_pio_handle_t pio_handle,
    udi_size_t scratch_requirement );
```

ARGUMENTS

pio_handle is a special PIO handle, previously acquired from `udi_pio_map`, whose associated trans list is designed to stop the associated device from initiating (mastering) transactions on its bus in case the driver fails.

scratch_requirement is the number of bytes of scratch space needed during processing of the associated trans list.

DESCRIPTION

When a driver does something illegal, causes a system fault, or otherwise misbehaves, the UDI environment can “kill” the corresponding driver region. However, since the faulting driver region can no longer be trusted to execute correctly, the environment needs some way to stop the corresponding device from proceeding without having to re-execute the faulting region. `udi_pio_abort_sequence` provides the environment a PIO handle (with an associated ***trans_list***) which it can use for this purpose.

The driver calls `udi_pio_abort_sequence` to register ***pio_handle*** with the environment. ***pio_handle*** must be mapped with a trans list that can be used to stop the corresponding device from initiating (mastering) transactions on the bus, including the generation of DMA transactions and interrupts. The PIO sequence specified by the associated trans list does not need to flush data or preserve device state, and should do the simplest sequence possible to stop the device, such as resetting it or otherwise stopping its ability to do bus mastering.

If needed, the registered trans list will be executed as if by `udi_pio_trans`. The `UDI_PIO_SCRATCH` addressing mode may be used to access up to ***scratch_requirement*** bytes of scratch space. `UDI_PIO_BUF` and `UDI_PIO_MEM` must not be used.

The abort trans list will be executed immediately in its own serialization domain without regard to the state of other PIO operations in other serialization domains; the abort operations will preempt any PIO trans lists currently executing or scheduled for executing and those PIO trans lists will be deallocated by the environment rather than being continued or executed. Additionally, no regions of this driver instance will be entered after initiation of the abort trans list and all channels to parents of that driver instance will be closed to release associated resources.

To facilitate handling faults in as wide a portion of the driver as possible, the driver should call `udi_pio_abort_sequence` as early as possible in its per-instance initialization sequence. If the device changes state in such a way that a different procedure is needed to shut it down, the driver may call `udi_pio_abort_sequence` again to replace the previously-registered sequence, but this should only be done if absolutely necessary.

The PIO handle passed to this service call is “given away” (as if with `udi_pio_unmap`). The driver must no longer access this handle.

A driver is not required to register a PIO abort sequence with this call if the operational characteristics of the device are such that it will not generate any activity (DMA, interrupts, etc) even if the driver is abruptly removed.

REFERENCES `udi_pio_map, udi_pio_unmap, udi_pio_trans_t`

4.3 PIO Access Service Calls and Structures

PIO devices are accessed by passing a list of PIO transactions to `udi_pio_trans`. Each of the transactions are processed in turn. When all the transactions in the list are completed, the driver's callback routine is called.

There are three types of PIO transactions that affect PIO devices: input, output, and synchronization. Input transactions read data from a device; output transactions send data to a device; synchronization transactions are used to control ordering of input/output transactions on weakly-ordered architectures.

The following rules and restrictions apply to all PIO transactions:

1. The ***tran_size*** parameter of each transaction descriptor must be from 0 to 5 (which may be represented as `UDI_PIO_1BYTE`, `UDI_PIO_2BYTE`, ...). Since it encodes size as a power of two, this represents transaction sizes from 1 to 32 bytes (256 bits). For transaction descriptors to which transaction size does not apply (e.g. `UDI_PIO_BRANCH`), ***tran_size*** must be zero.
2. Endian translation is performed, when needed, on each `2^tran_size` byte quantity. To transfer a character or byte array to or from device memory the driver must use a ***tran_size*** of `UDI_PIO_1BYTE` with an appropriate repeat count rather than using `UDI_PIO_NEVERSWAP` with a single basic transaction.
3. If the ***pio_attributes*** parameter to `udi_pio_map` did not include `UDI_PIO_UNALIGNED`, then the ***base_offset*** in `udi_pio_map`, the base address alignment of the register set itself, and any PIO offsets accessed by PIO transactions must all be transaction-size aligned (i.e. multiples of `2^tran_size`).
4. If copied to driver memory, the datum must be transaction-size aligned relative to a `udi_mem_alloc`'d piece of memory, a control block's scratch area, or a variable whose data type is `2^tran_size` bytes in size, even if `UDI_PIO_UNALIGNED` was set in ***pio_attributes***.
5. PIO pacing, when indicated, occurs between each `2^tran_size` byte transfer that accesses the PIO device (once per repetition). The only PIO transaction types that access the PIO device are `UDI_PIO_IN`, `UDI_PIO_OUT`, `UDI_PIO_IN_IND`, `UDI_PIO_OUT_IND`, `UDI_PIO_REP_IN_IND`, and `UDI_PIO_REP_OUT_IND`.
6. If a PIO device access uses a transaction size that is one of the atomic transaction sizes indicated by `udi_pio_atomic_sizes`, all `2^tran_size` bytes are guaranteed to be transferred together "atomically" (as defined in "atomic transaction" in the Glossary), unless `UDI_PIO_UNALIGNED` was set in ***pio_attributes***.

Since odd sizes (eg. 3 bytes) are not allowed, access to, for example, a 24-bit register must either be encapsulated within a larger access and the unused bits appropriately tossed (as described below), or it must be split up into smaller accesses (eg. 1- and 2-byte accesses) as appropriate for the IO card or shared control structure.

For example, a 24-bit register at PIO offsets 1 through 3 in little endian device memory can be read in an endian-neutral manner by doing a 32-bit read at PIO offset 0 and shifting the result right by 8 bits.

A 24-bit register at offsets 0 through 2 could be read by doing the 32-bit read and then extracting the low-order 3 bytes using a mask of 0xFFFFFF.

A 24-bit register at offsets 2 through 4 could be read by doing a 16-bit read at PIO offset 2, and adding this to the result of an 8-bit read at offset 4 shifted left by 16 bits.

NAME **udi_pio_trans_t** *PIO transaction descriptor*

SYNOPSIS

```
#include <udi.h>
#include <udi_physio.h>

typedef const struct {
     udi_ubit8_t pio_op;
     udi_ubit8_t tran_size;
     udi_ubit16_t operand;
} udi_pio_trans_t;

/* Values for tran_size */
#define UDI_PIO_1BYTE                      0
#define UDI_PIO_2BYTE                      1
#define UDI_PIO_4BYTE                      2
#define UDI_PIO_8BYTE                      3
#define UDI_PIO_16BYTE                     4
#define UDI_PIO_32BYTE                     5

/* Values for register numbers in pio_op */
#define UDI_PIO_R0                         0
#define UDI_PIO_R1                         1
#define UDI_PIO_R2                         2
#define UDI_PIO_R3                         3
#define UDI_PIO_R4                         4
#define UDI_PIO_R5                         5
#define UDI_PIO_R6                         6
#define UDI_PIO_R7                         7

/* Values for addressing modes in pio_op */
#define UDI_PIO_DIRECT                     0x00
#define UDI_PIO_SCRATCH                    0x08
#define UDI_PIO_BUF                        0x10
#define UDI_PIO_MEM                        0x18

/* Values for Class A opcodes in pio_op */
#define UDI_PIO_IN                         0x00
#define UDI_PIO_OUT                        0x20
#define UDI_PIO_LOAD                       0x40
#define UDI_PIO_STORE                      0x60

/* Values for Class B opcodes in pio_op */
#define UDI_PIO_LOAD_IMM                   0x80
#define UDI_PIO_CSKIP                      0x88
#define UDI_PIO_IN_IND                     0x90
#define UDI_PIO_OUT_IND                    0x98
#define UDI_PIO_SHIFT_LEFT                 0xA0
#define UDI_PIO_SHIFT_RIGHT                0xA8
#define UDI_PIO_AND                        0xB0
#define UDI_PIO_AND_IMM                    0xB8
```

```
#define UDI_PIO_OR                      0xC0
#define UDI_PIO_OR_IMM                  0xC8
#define UDI_PIO_XOR                     0xD0
#define UDI_PIO_ADD                     0xD8
#define UDI_PIO_ADD_IMM                 0xE0
#define UDI_PIO_SUB                     0xE8

/* Values for Class C opcodes in pio_op */
#define UDI_PIO_BRANCH                  0xF0
#define UDI_PIO_LABEL                   0xF1
#define UDI_PIO_REP_IN_IND              0xF2
#define UDI_PIO_REP_OUT_IND             0xF3
#define UDI_PIO_DELAY                   0xF4
#define UDI_PIO_BARRIER                 0xF5
#define UDI_PIO_SYNC                    0xF6
#define UDI_PIO_SYNC_OUT                0xF7
#define UDI_PIO_DEBUG                   0xF8
#define UDI_PIO_END                     0xFE
#define UDI_PIO_END_IMM                 0xFF


/* Values for UDI_PIO_DEBUG operand */
#define UDI_PIO_TRACE_OPS_NONE          0
#define UDI_PIO_TRACE_OPS1              1
#define UDI_PIO_TRACE_OPS2              2
#define UDI_PIO_TRACE_OPS3              3
#define UDI_PIO_TRACE_REGS_NONE         (0U<<2)
#define UDI_PIO_TRACE_REGS1             (1U<<2)
#define UDI_PIO_TRACE_REGS2             (2U<<2)
#define UDI_PIO_TRACE_REGS3             (3U<<2)
#define UDI_PIO_TRACE_DEV_NONE          (0U<<4)
#define UDI_PIO_TRACE_DEV1              (1U<<4)
#define UDI_PIO_TRACE_DEV2              (2U<<4)
#define UDI_PIO_TRACE_DEV3              (3U<<4)
```

MEMBERS

pio_op is the type of operation to use for this transaction. ***pio_op*** encodes the type of transaction and the selection of source and destination operands.

tran_size is the power-of-2 size of a basic PIO transaction. The actual size is 2^{tran_size} bytes. The ***tran_size*** value must be from 0 to 5, corresponding to a transaction size of 1 to 32 bytes (256 bits). The mnemonic constants `UDI_PIO_1BYTE` through `UDI_PIO_32BYTE` are available for use in setting ***tran_size***.

operand is an operand value or address for the transaction. Its interpretation depends on the type of operation given by ***pio_op***.

DESCRIPTION

A PIO transaction descriptor (`udi_pio_trans_t`) describes in most cases a single PIO register/memory access transaction or an operation using a set of temporary registers. Multiple such transaction descriptors may be combined into a single list, called a *trans list*, to perform more complex sequences with a single call to `udi_pio_trans`. Advanced operations in the trans list allow for bit manipulations, repeat counts, serialization, and conditional branching. All PIO transactions are subject to the rules and procedural information listed on page 4-13.

Each element in the trans list array represents a basic transaction, and operates on a single transaction-sized piece of data. When transferring data to or from a device (for example, with UDI_PIO_IN or UDI_PIO_OUT), the entire transaction-size bytes will be transferred as a single atomic bus transaction if possible. See **udi_pio_atomic_sizes** on page 4-10 for a description of how to determine the atomic sizes supported on a particular platform.

While simple transactions can be handled with simple operations, UDI supports a rich set of PIO operation types, so that even fairly sophisticated manipulation of device data can be performed with a single call to `udi_pio_trans`. To allow these operations to be efficiently expressed, `udi_pio_trans` uses an abstract register load/store model.

During execution of `udi_pio_trans`, the UDI environment maintains a set of 8 temporary “registers”, numbered 0 through 7, each of which can hold one data item up to 32 bytes (256 bits) in size. Most PIO operations use one or more of these registers. If the size of a value loaded into a register (determined by ***tran_size***) is less than 32 bytes, the “upper” (most-significant) bytes are treated as zero if the register value is subsequently used with a larger transaction size. If a register value is used in an operation with a smaller transaction size than when the register was last loaded, the upper bytes will be ignored. Arithmetic operations do not generate underflows or overflows, they simply wrap around; in other words, all arithmetic is modulo `2^((2^`***tran_size***`)*8)`.

In addition to using the temporary registers, some PIO operations allow access to permanent memory associated with one of the arguments to `udi_pio_trans`. These operations can access control block scratch space, `udi_buf_t` buffer contents, or driver memory. Values in permanent memory are stored in the driver’s endianness, even for transaction sizes larger than native word sizes. These values will be either purely little endian or purely big endian. (See the definitions of **big endian** and **little endian** in Section 3.2.2, “Common Terms,” on page 3-2 of the UDI Core Specification.)

PIO operations are divided into 3 classes, based on the type of operands used: Class A, Class B, and Class C operations.

Class A Operations: Class A operations can use register values or permanent memory values. The ***pio_op*** value for a class A operation selects a register number (0-7) as well as an addressing mode from the following table.

Table 4-1 PIO Addressing Modes

Mnemonic	Value	Description
UDI_PIO_DIRECT	0x00	Register contents are used directly
UDI_PIO_SCRATCH	0x08	Register holds 32-bit offset into scratch space
UDI_PIO_BUF	0x10	Register holds 32-bit offset into buffer data
UDI_PIO_MEM	0x18	Register holds 32-bit offset into memory block

To set ***pio_op*** for a Class A operation, the register number and exactly one of the addressing mode mnemonics must be added to the opcode mnemonic:

pio_op = operation_code + addressing_mode + register

The mnemonic constants `UDI_PIO_R0` through `UDI_PIO_R7` are available for use in selecting registers for all opcodes and operands that use register numbers.

Class A ***pio_op*** values are encoded with the following bit patterns: `000aarrr - 011aarrr`, where `aa` selects the addressing mode according to Table 4-1 and `rrr` selects a register number.

Class B Operations: Class B operations can use register values but not permanent memory values. The ***pio_op*** value for a class B operation selects a register number (0-7), which is used as if the UDI_PIO_DIRECT addressing mode were specified.

To set ***pio_op*** for a Class B operation, the register number must be added to the opcode mnemonic:

pio_op = operation_code + register

Class B ***pio_op*** values are encoded with the following bit patterns: `10000rrr - 11101rrr`, where `rrr` selects a register number to be used with that addressing mode.

Class C Operations: Class C operations use neither register values nor permanent memory values. The ***pio_op*** value for a class C operation consists only of the opcode:

pio_op = operation_code

Class C ***pio_op*** values are encoded with the following bit patterns: `11110000 - 11111111`.

The following tables list the opcodes for each class and their encoded values, along with the corresponding operation definitions, including an indication of how the ***operand*** value is used in each case. In these tables, *addr* means the contents of the location referred to by the selected addressing mode and register, `reg(operand)` means a direct register value where the low-order 3 bits of ***operand*** are used to select the register number (unless otherwise specified, the remaining bits must be zero), `PIO(operand)` means a location on the PIO device where ***operand*** indicates the desired PIO offset, and `reg` means the selected register contents.

Table 4-2 Class A PIO Operation Codes

Operation Code	**Value**	**Operation**
UDI_PIO_IN	0x00	*addr* `<- PIO(operand)`
UDI_PIO_OUT	0x20	`PIO(operand) <-` *addr*
UDI_PIO_LOAD	0x40	`reg(operand) <-` *addr*
UDI_PIO_STORE	0x60	*addr* `<- reg(operand)`

Table 4-3 Class B PIO Operation Codes

Operation Code	**Value**	**Operation**
UDI_PIO_LOAD_IMM	0x80	`reg <- operand` (multi-part)
UDI_PIO_CSKIP	0x88	Conditional skip. The value in `reg` is compared with zero. The `operand` value selects a condiction code from Table 4-5 to determine the type of comparison. If the condition is TRUE, the following trans list element will be skipped.
UDI_PIO_IN_IND	0x90	`reg <- PIO(reg(operand))`
UDI_PIO_OUT_IND	0x98	`PIO(reg(operand)) <- reg`
UDI_PIO_SHIFT_LEFT	0xA0	`reg <- reg << operand`
UDI_PIO_SHIFT_RIGHT	0xA8	`reg <- reg >> operand`
UDI_PIO_AND	0xB0	`reg <- reg & reg(operand)`
UDI_PIO_AND_IMM	0xB8	`reg <- reg & operand` (zero-extended)
UDI_PIO_OR	0xC0	`reg <- reg | reg(operand)`
UDI_PIO_OR_IMM	0xC8	`reg <- reg | operand` (zero-extended)
UDI_PIO_XOR	0xD0	`reg <- reg ^ reg(operand)`
UDI_PIO_ADD	0xD8	`reg <- reg + reg(operand)`
UDI_PIO_ADD_IMM	0xE0	`reg <- reg + operand` (sign-extended)
UDI_PIO_SUB	0xE8	`reg <- reg - reg(operand)`

Table 4-4 Class C PIO Operation Codes

Operation Code	Value	Operation
UDI_PIO_BRANCH	0xF0	Unconditional branch to the UDI_PIO_LABEL with the same ***operand*** value. ***tran_size*** is unused and must be set to zero.
UDI_PIO_LABEL	0xF1	Destination of a UDI_PIO_BRANCH. ***tran_size*** is unused and must be set to zero.
UDI_PIO_REP_IN_IND	0xF2	Repeated indirect PIO input transactions. ***operand*** is encoded according to the UDI_PIO_REP_ARGS macro described below.
UDI_PIO_REP_OUT_IND	0xF3	Repeated indirect PIO output transactions. ***operand*** is encoded according to the UDI_PIO_REP_ARGS macro described below.
UDI_PIO_DELAY	0xF4	Delay for at least ***operand*** microseconds during a PIO wait loop.
UDI_PIO_BARRIER	0xF5	Place an ordering barrier between PIO transactions. ***tran_size*** is unused and must be set to zero. See below for details on the behavior of this operation and its use of ***operand***.
UDI_PIO_SYNC	0xF6	Synchronize with respect to outstanding PIO transactions. See below for details on the behavior of this operation and its use of ***tran_size*** and ***operand***.
UDI_PIO_SYNC_OUT	0xF7	Synchronize with respect to outstanding PIO output transactions. See below for details on the behavior of this operation and its use of ***tran_size*** and ***operand***.
UDI_PIO_DEBUG	0xF8	Enable/disable debug tracing of PIO transactions.
UDI_PIO_END	0xFE	Terminate processing. `result_code <- reg(operand)` ***tran_size*** must be <= `UDI_PIO_2BYTE`.
UDI_PIO_END_IMM	0xFF	Terminate processing. `result_code <- operand` ***tran_size*** must be `UDI_PIO_2BYTE`.

Detailed description of Addressing Modes:

UDI_PIO_DIRECT — In this mode, the contents of the selected register, `reg`, are used for the *`addr`* value itself. In this case, *`addr`* is unaffected by any stride values from `UDI_PIO_REP_IN_IND` or `UDI_PIO_REP_OUT_IND` operations.

UDI_PIO_SCRATCH — In this mode, the contents of the selected register, `reg`, are used as an offset into the scratch space of the control block passed to `udi_pio_trans`. The scratch space data bytes starting at this offset and extending for the selected transaction size are used as the *`addr`* value.

Only the low-order 32 bits of the register value are used; any higher order bits are ignored; smaller values previously loaded into the register are zero-extended to 32 bits. The offset must be a multiple of the transaction size.

UDI_PIO_BUF — In this mode, the contents of the selected register, `reg`, are used as an offset into the valid data area of the `udi_buf_t` buffer passed to `udi_pio_trans`. The buffer data bytes starting at this offset and extending for the selected transaction size are used as the ***`addr`*** value. Only the low-order 32 bits of the register value are used; any higher order bits are ignored; smaller values previously loaded into the register are zero-extended to 32 bits. The offset must be a multiple of the transaction size. All offsets accessed with UDI_PIO_BUF must be less than the buffer's ***`buf_size`*** value at the time `udi_pio_trans` was called.

UDI_PIO_MEM — In this mode, the contents of the selected register, `reg`, are used as an offset into the auxiliary memory block pointed to by the ***`mem_ptr`*** argument passed to `udi_pio_trans`. The memory block data bytes starting at this offset and extending for the selected transaction size are used as the ***`addr`*** value. Only the low-order 32 bits of the register value are used; any higher order bits are ignored; smaller values previously loaded into the register are zero-extended to 32 bits. The offset must be a multiple of the transaction size.

When PIO device registers/memory are used as the source (UDI_PIO_IN, UDI_PIO_IN_IND, or UDI_PIO_REP_IN_IND) or destination (UDI_PIO_OUT, UDI_PIO_OUT_IND, or UDI_PIO_REP_OUT_IND) of an operation, the specified PIO offset is interpreted relative to the register set and base offset specified through arguments to `udi_pio_map` and passed to `udi_pio_trans` via the ***`pio_handle`*** argument. UDI_PIO_IN and UDI_PIO_OUT only support PIO offsets up to 65535. UDI_PIO_IN_IND, UDI_PIO_OUT_IND, UDI_PIO_REP_IN_IND and UDI_PIO_REP_OUT_IND use a register value to provide the PIO offset, allowing for indirections and larger offsets; the 32 low-order bits of the register value are used for the PIO offset; any higher order bits are ignored; smaller values previously loaded into the register are zero extended to 32 bits.

UDI_PIO_LOAD_IMM:

The immediate load operation takes a value directly from the ***`operand`*** portion of a transaction descriptor and loads it into the selected temporary register. The transaction size for this operation must be at least 2 bytes.

If `UDI_PIO_LOAD_IMM` is used with a transaction size greater than 2 bytes, multiple trans list elements are used to hold all the bytes of the immediate value. Starting from the least significant 2 bytes, pairs of bytes are loaded from each successive ***`operand`*** value starting from the `UDI_PIO_LOAD_IMM` trans list entry, where each pair is listed as most-significant-byte followed by least-significant-byte. The subsequent trans list elements which contain the additional byte values must repeat the ***pio_op*** and ***tran_size*** values from the initial `UDI_PIO_LOAD_IMM` operation. Normal trans list execution resumes after the last such element.

Shift operations:

UDI_PIO_SHIFT_LEFT and UDI_PIO_SHIFT_RIGHT use the ***operand*** value as a shift count for shifting the entire transaction-size byte register value. The shift count must be from 1 to 32. Any bits shifted out of the low order transaction-size bytes will be discarded. All bits shifted in to the low order transaction-size bytes will be zero. Any previous more significant bytes shall effectively be zeroed.

Repeat operations:

A UDI_PIO_REP_IN_IND or UDI_PIO_REP_OUT_IND operation can be used to repeat a basic PIO transaction up to 2^{32}-1 times. The basic PIO transaction is like a UDI_PIO_IN_IND or UDI_PIO_OUT_IND, in that it performs a PIO transaction between a PIO offset indicated in one register and a permanent memory location addressed via another register. A third register holds the repeat count, which must be from zero to 2^{32}-1.

The ***operand*** value for a repeat operation holds a number of parameters, intialized via the UDI_PIO_REP_ARGS macro. See **UDI_PIO_REP_ARGS** on page 4-26 for more details on repeat operations.

Branching operations:

UDI_PIO_BRANCH is an unconditional branch operation. The ***operand*** value is used to find the next trans list element to execute instead of continuing in order. The operand value is matched against UDI_PIO_LABEL trans list elements; execution continues after the UDI_PIO_LABEL with the same ***operand*** value as the UDI_PIO_BRANCH. It is illegal to have two UDI_PIO_LABEL elements with the same ***operand*** value in one trans list. It is illegal to have a UDI_PIO_BRANCH that does not have a corresponding UDI_PIO_LABEL.

It is illegal to specify a `UDI_PIO_LABEL` or `UDI_PIO_BRANCH` transaction with an ***operand*** value of zero.

If a UDI_PIO_LABEL operation is executed sequentially, it acts as a no-op.

Conditional operation:

UDI_PIO_CSKIP is a conditional skip operation. It compares a register value with zero and skips the following operation if the comparison meets the required conditions. The ***operand*** specifies the condition code, according to the following table:

Table 4-5 PIO Condition Codes

Mnemonic	**Value**	**Condition Description**
UDI_PIO_Z	`0`	`reg == 0`
UDI_PIO_NZ	`1`	`reg != 0`
UDI_PIO_NEG	`2`	`reg < 0` [signed]
UDI_PIO_NNEG	`3`	`reg >= 0` [signed]

Busy-wait operation:

A UDI_PIO_DELAY pauses execution of the trans list for a specified time. Transaction lists that loop waiting for device events (such as ready bits being set in status registers) must include a UDI_PIO_DELAY in the loop to avoid consuming excessive CPU resources. The ***operand*** value provides the desired delay time in microseconds. This is a **hint** to the environment, and must not be used to perform precise timings. The environment will pause the execution of the PIO trans list for *at least* the requested amount of time.

Synchronization operations:

The synchronization operations (UDI_PIO_BARRIER, UDI_PIO_SYNC and UDI_PIO_SYNC_OUT) do not actually transfer any new data. Instead, they make sure that any PIO transactions that have already been generated are made visible to the device before any subsequent PIO transactions to/from the same device register set become visible (thus acting as a "barrier"). These operations only affect accesses to the device or memory referenced by the PIO handle, not buffer or driver memory.

The ***operand*** value for UDI_PIO_BARRIER must be either 0 or UDI_PIO_OUT. If UDI_PIO_OUT, the barrier is only guaranteed to act with respect to device output transactions; this may be faster than a full barrier.

The UDI_PIO_SYNC and UDI_PIO_SYNC_OUT operations provide even stronger synchronization: they also wait for the affected transactions to reach the device. UDI_PIO_SYNC affects all PIO transactions; UDI_PIO_SYNC_OUT is only required to affect output transactions. ***operand*** and ***tran_size*** must be set to a PIO offset range of the device that causes no side effects if input from. On some platforms, PIO transactions can only be synchronized by performing an additional PIO input transaction. In such cases, the environment will read ***2**$^{tran\_size}$* bytes from offset ***operand*** of the device (and discard the result).

Synchronization operations are not needed if UDI_PIO_STRICTORDER was specified in the PIO attributes for the PIO handle. With UDI_PIO_STRICTORDER, each basic transaction that accesses the PIO device is followed by an implicit UDI_PIO_SYNC.

Upon completion of a `udi_pio_trans` sequence, via execution of a UDI_PIO_END operation, an implicit UDI_PIO_BARRIER with ***operand*** of zero is performed.

Environment implementations may choose to impose stronger ordering and synchronization than required by the synchronization operations used, up to and including strict ordering.

Debugging Operation:

The UDI_PIO_DEBUG operation controls PIO debug tracing. This opcode will be seldom, if ever, used in production drivers and is to be used mostly while a driver is under development. As with the other debugging facilities in UDI like `udi_debug_printf` and `udi_debug_break`, an environment may choose to completely or partially ignore this opcode. Each UDI environment must document how the debugging output is made available to the developer.

Upon entry to a trans list, all debugging is disabled. UDI_PIO_DEBUG opcodes are interpreted like all other opcodes; they are synchronous with the execution of the trans list. Each UDI_PIO_DEBUG opcode will reset the debug level regardless of the current debug level; they are not cumulative.

The ***trans_size*** must be zero since no PIO-visible data is transferred by this opcode. The operand provides a bitmask to control the level of debugging. If no flags are specified, debugging is disabled.

For each field, tracing information is generated only if the current debug level in the trans list is greater than or equal to the debugging level of the environment. The output at various levels is explictly specified; it's an agreement between the trans list author and the user of the environment controlling the trace level.

Termination Operations:

Upon reaching a UDI_PIO_END_IMM, processing of the trans list is terminated and the result code (an arbitrary 16-bit code that is defined by the driver) in the `udi_pio_trans_call_t` callback is set to the low byte of the ***`operand`*** value.

Upon reaching a UDI_PIO_END, processing of the trans list terminates as in the UDI_PIO_END_IMM case, except that the ***`operand`*** specifies the register number whose contents are to be returned as the result code.

The last element of a PIO trans list must be either a UDI_PIO_END operation, a UDI_PIO_END_IMM operation, or a UDI_PIO_BRANCH.

REFERENCES `udi_pio_map, udi_pio_trans, UDI_PIO_REP_ARGS, udi_buf_t, udi_mem_alloc`

NAME

UDI_PIO_REP_ARGS *Parameters for repeated PIO transactions*

SYNOPSIS

```
#include <udi.h>
#include <udi_physio.h>

#define \
   UDI_PIO_REP_ARGS ( \
     mode, mem_reg, mem_stride, \
     pio_reg, pio_stride, cnt_reg ) \
          ((mode)|(mem_reg)|((mem_stride)<<5)| \
           ((pio_reg)<<7)|((pio_stride)<<10)| \
           ((cnt_reg)<<13))
```

ARGUMENTS

mode is the addressing mode used to access the memory operand, according to Table 4-1, "PIO Addressing Modes," on page 4-18.

mem_reg is the number of the register used to access the memory operand, according to Table 4-1.

mem_stride is the stride value used to increment the memory offset between repeats.

pio_reg is the number of the register used to hold the initial PIO offset. The 32 low-order bits of the register value are used for the PIO offset; any higher order bits are ignored; smaller values previously loaded into the register are zero extended to 32 bits.

pio_stride is the stride value used to increment the PIO offset between repeats.

cnt_reg is the number of the register used to hold the repeat count. The 32 low-order bits of the register value are used for the count; any higher order bits are ignored; smaller values previously loaded into the register are zero extended to 32 bits.

DESCRIPTION

The `UDI_PIO_REP_ARGS` macro is used to construct the ***operand*** value for `UDI_PIO_REP_IN_IND` and `UDI_PIO_REP_OUT_IND` repeating PIO operations (see **udi_pio_trans_t** on page 4-15).

A repeat operation repeats a basic PIO transaction the number of times indicated by the repeat count from the ***cnt_reg*** register. The memory location and PIO offset for the first repetition are determined by ***mode***, ***mem_reg***, and ***pio_reg***. For subsequent repetitions, the memory offset (if ***mode*** is not `UDI_PIO_DIRECT`) and PIO offset are incremented according to the corresponding stride values. The values in the original registers (***mem_reg***, ***pio_reg***, and ***cnt_reg***) are not affected by stride increments or by repeat count decrements.

The stride values ***`mem_stride`*** and ***`pio_stride`*** indicate a (possibly zero) multiple of the target operation's transaction size, according to the following table.

Table 4-6 Stride Values for PIO Repeat Operations

Stride Code	Stride Size in Bytes	Multiple of Transaction Size
`0`	`0`	`0`
`1`	`2^`***`tran_size`***	`1`
`2`	`2^(`***`tran_size`***`+1)`	`2`
`3`	`2^(`***`tran_size`***`+2)`	`4`

Some examples of how to use stride values are shown in the following table.

Table 4-7 Example Uses of PIO Stride Parameters

Function	mem_stride	pio_stride
Copy array to/from a contiguous range of device memory	`1`	`1`
Copy array to/from single device register	`1`	`0`
Fill range of device memory w/single value	`0`	`1`
Fill lower 2 bytes of a series of 4-byte aligned device registers (*tran_size* = 2) with a single value	`0`	`2`
Fill lower 2 bytes of a series of 4-byte aligned device registers (*tran_size* = 2) from an array of 2-byte values	`1`	`2`

NAME

udi_pio_trans *Generate PIO transactions*

SYNOPSIS

```
#include <udi.h>
#include <udi_physio.h>

void udi_pio_trans (
     udi_pio_trans_call_t *callback,
     udi_cb_t *gcb,
     udi_pio_handle_t pio_handle,
     udi_index_t start_label,
     udi_buf_t *buf,
     void *mem_ptr );

typedef void udi_pio_trans_call_t (
     udi_cb_t *gcb,
     udi_buf_t *new_buf,
     udi_status_t status,
     udi_ubit16_t result );
```

ARGUMENTS

callback, gcb are standard arguments described in the "Asynchronous Service Calls" section of *"Standard Calling Sequences"* in the UDI Core Specification.

pio_handle is the PIO handle associated with the desired device register set and transaction list, previously acquired from `udi_pio_map`.

start_label specifies which `UDI_PIO_LABEL` value the trans list execution should begin at. A ***start_label*** value of zero indicates that the trans list execution should start from the beginning, otherwise values from 1-7 indicate a request to start the trans list at the corresponding `UDI_PIO_LABEL` position. Any other values for this argument are illegal.

buf is a pointer to a data buffer that can be accessed by the PIO transactions using UDI_PIO_BUF. If ***buf*** is set to NULL, UDI_PIO_BUF transactions are illegal for this call.

mem_ptr is a pointer to an auxiliary memory block that can be accessed by PIO transactions that specify UDI_PIO_MEM. If ***mem_ptr*** is set to NULL, such transactions are illegal for this call.

new_buf is a possibly new UDI buffer that the driver must use in place of the original ***buf*** after the callback is called, since the environment may need to re-allocate the buffer to accomodate new data added to the buffer as a result of this service call.

status is a UDI status code. If the PIO transactions all completed without error, it will be set to UDI_OK; otherwise it will be set to `UDI_STAT_HW_PROBLEM`. If not UDI_OK, the status code may already contain a correlate value.

result is a result code set by a UDI_PIO_END or a UDI_PIO_END_IMM transaction.

DESCRIPTION

udi_pio_trans performs a series of one or more PIO transactions to the area of a device indicated by the PIO handle, ***pio_handle***. The list of (possibly repeated) transactions to perform is given by the ***trans_list*** which was associated with the PIO handle at the time of udi_pio_map. See **udi_pio_trans_t** on page 4-15 for a detailed description of PIO transactions.

Synchronization of udi_pio_trans calls and the execution of the corresponding transaction lists is provided by the environment independently from the region synchronization. For proper driver design refer to the PIO synchronization and ordering definitions on page 4-1.

If the PIO attributes associated with ***pio_handle*** indicate that the endianness of the device's data structures are different from the driver's endianness, udi_pio_trans will automatically perform necessary byte-swapping. The driver must not do the byte-swapping itself, as udi_pio_trans may be able to take advantage of special hardware support for byte-swapped PIO transactions.

Pacing delays and other attributes encoded in ***pio_handle***, such as ordering and atomicity requirements, will be applied to each individual I/O operation.

Once the last transaction has been executed, including any pacing delay, the callback routine will be called. An implicit UDI_PIO_BARRIER will be executed at the end of the transaction list, but if UDI_PIO_SYNC or UDI_PIO_SYNC_OUT are required they must be explicitly included.

WARNINGS

Use of the ***mem_ptr*** parameter must conform to the rules described in "Using Memory Pointers with Asynchronous Service Calls" on page 5-2 of the UDI Core Specification.

STATUS VALUES

A UDI status code indicating the success or failure of the PIO transactions. If a hardware error, such as a bus timeout or parity error, is detected in conjunction with a PIO transaction, execution of the PIO transaction list is terminated and status is set to UDI_STAT_HW_PROBLEM. Transactions following the one that caused the error may or may not be executed. Environments are not required to detect errors during udi_pio_trans; in such environments, the effect of a PIO hardware error is indeterminate and may include driver or system termination. If a driver expects that the PIO might reasonably fail (e.g. because the device is not present) it must use udi_pio_probe instead of udi_pio_trans.

REFERENCES

udi_pio_map, udi_pio_trans_t, udi_pio_probe

NAME

udi_pio_probe *Probe a PIO device that might not be present*

SYNOPSIS

```
#include <udi.h>
#include <udi_physio.h>

void udi_pio_probe (
     udi_pio_probe_call_t *callback,
     udi_cb_t *gcb,
     udi_pio_handle_t pio_handle,
     void *mem_ptr,
     udi_ubit32_t pio_offset,
     udi_ubit8_t tran_size,
     udi_ubit8_t direction );

typedef void udi_pio_probe_call_t (
     udi_cb_t *gcb,
     udi_status_t status );

/* Values for direction */
#define UDI_PIO_IN                        0x00
#define UDI_PIO_OUT                       0x20
```

ARGUMENTS

callback, gcb are standard arguments described in the "Asynchronous Service Calls" section of *"Standard Calling Sequences"* in the UDI Core Specification.

pio_handle is a PIO handle previously allocated via `udi_pio_map`.

mem_ptr is a pointer to a memory location which is the source or destination of the PIO transaction.

pio_offset is the offset into the mapped device at which to perform the transaction. There are no alignment requirements on `pio_offset`.

tran_size is the power-of-2 size of the PIO transaction. The actual size is 2^{tran_size} bytes. The ***tran_size*** value must be from 0 to 5, corresponding to a transaction size of 1 to 32 bytes (256 bits). The mnemonic constants `UDI_PIO_1BYTE` through `UDI_PIO_32BYTE` are available for use in setting ***tran_size***.

direction is exactly one of `UDI_PIO_IN` or `UDI_PIO_OUT`, to indicate the direction of transfer.

status indicates the success or failure of the PIO probe attempt.

DESCRIPTION

This service call attempts to perform a PIO access to a device that may or may not be present. It will do this in such a way that there will be no adverse side effects (such as driver or system aborts) if there is in fact no device present. If a different device is present at the same or overlapping location then it might be accessed instead of the intended device, causing an indeterminate effect on the state of that device; environments are not required to prevent this from happening.

The trans list associated with ***pio_handle*** is ignored; instead, a single basic PIO transaction of 2^{tran_size} bytes is performed, in the direction indicated by ***direction***.

If a driver's device might not have been reliably enumerated and might not be actually present, `udi_pio_probe` must be used to determine if the device is present before using it for normal operation. Otherwise, `udi_pio_probe` should not be used. It is likely to be slower than `udi_pio_trans`, possibly affecting overall system performance as well. In some environments, use of `udi_pio_probe` may force the driver to be run only during certain stages of system initialization; such an environment may require system re-initialization in order to use a driver that uses `udi_pio_probe`.

In order for a driver to use `udi_pio_probe`, its region's "`pio_probe`" region attribute must be set to "`yes`". Some environments may refuse to use drivers with "`pio_probe`" set to "`yes`". See Section 1.4, "Extensions to Static Driver Properties," on page 1-2 for details.

Transactions using `udi_pio_probe` are not guaranteed to be atomic.

If a driver determines that its device is not present during binding, it must log an error using `udi_log_write` and respond to the Management Agent with a status of `UDI_STAT_NOT_RESPONDING` via `udi_channel_event_complete`.

WARNINGS

Use of the ***mem_ptr*** parameter must conform to the rules described in "Using Memory Pointers with Asynchronous Service Calls" on page 5-2 of the UDI Core Specification.

STATUS VALUES

UDI_OK

UDI_STAT_HW_PROBLEM – the device failed to respond to the PIO access. Some environments may return "garbage" data in such cases, rather than indicating an error with `UDI_STAT_HW_PROBLEM`.

REFERENCES

`udi_pio_trans`

4.4 PIO Handle Transferability

Since PIO handles are transferable between regions, they need to be represented in layout specifiers for use by metalanguages and by drivers that specify inline layouts. This section defines an extension to the data layout specifier defined in the UDI Core Specification.

NAME

udi_layout_t (PIO) *Data layout specifier for PIO*

SYNOPSIS

```
#include <udi.h>
#include <udi_physio.h>

typedef udi_ubit8_t udi_layout_t;

/* PIO Handle Layout Element Type Code */
#define UDI_DL_PIO_HANDLE_T              200
```

DESCRIPTION

This page lists additional layout specifier codes that can be used with the `udi_layout_t` type, defined in the UDI Core Specification, to specify PIO-related data layouts.

A data layout specifier consists of an array of one or more `udi_layout_t` layout elements. Each element contains a type code indicating one of the UDI data types that can be passed into a channel operation, either as a field in the control block or as an additional parameter. Each successive element of the array represents successive offsets within the described structure, with padding automatically inserted for alignment purposes as if the specified data types had appeared in a C `struct` declaration.

A `UDI_DL_PIO_HANDLE_T` layout element represents a PIO handle, of type `udi_pio_handle_t`, which may be `UDI_NULL_PIO_HANDLE`.

REFERENCES

`udi_layout_t`

UDI Physical I/O Specification

Section 2: Bus Bridge Metalanguage

Bus Bridge Metalanguage 5

5.1 Overview

This chapter details the channel operations and their parameters for the Bus Bridge Metalanguage, which allow a driver for a direct-attached hardware device to communicate with the driver for its parent bus bridge. The device may be a leaf device or another bridge. For purposes of discussion we always refer to the child (furthest from the processor) as the device and the parent as the bridge, and we'll refer to the device driver and the bridge driver.

Each subsection defines the channel operation calls, control block type declarations, the rationale for the operation's existence, constraints and guidelines for the use of each operation, and error conditions that can occur. (The common errors described earlier are not repeated.)

The Bus Bridge Metalanguage operations are grouped into four roles. For each role there's a corresponding channel ops vector, which is registered via an associated udi_ops_init_t structure. The Bus Bridge Metalanguage bridge role supports binding/unbinding and interrupt registration operations invoked on a parent bridge object by a child device driver. The Bus Bridge Metalanguage device role supports binding/unbinding and interrupt registration operations invoked on a child device object by a parent bridge driver. The Bus Bridge Metalanguage interrupt dispatcher role supports interrupt acknowledgment operations invoked on a parent bridge object by a child device driver. The Bus Bridge Metalanguage interrupt handler role supports interrupt event operations invoked on a child device object by a parent bridge driver.

In the Bus Bridge Metalanguage, since each control block type is designed to be used across a group of related operations, a separate control block group is defined per individual type of control block. See **udi_cb_init_t** on page 10-11 of the UDI Core Specification for additional information.

Only the `udi_intr_event_rdy` Bus Bridge Metalanguage operation is abortable with `udi_channel_op_abort`. None of the Bridge Metalanguage operations are recoverable.

5.1.1 Versioning

All functions and structures defined in this chapter are part of the "`udi_bridge`" interface, currently at version "`0x101`". A driver that conforms to and uses the Bus Bridge Metalanguage of the UDI Physical I/O Specification, Version 1.01, must include the following declaration in its `udiprops.txt` file (see Chapter 30, *"Static Driver Properties"*, of the UDI Core Specification):

```
requires udi_bridge 0x101
```

Compile-time versioning and header files for the Bus Bridge Metalanguage are covered by the general requirements for the UDI Physical I/O Specification defined in Section 1.2, "General Requirements".

A portable implementation of the Bus Bridge Metalanguage must include a corresponding "provides" declaration in its `udiprops.txt` file, must conform to the same compile-time versioning and header requirements as for drivers, and must conform to the requirements specified in the Metalanguage-to-Environment (MEI) interface defined in Chapter 27, *"Introduction to MEI"*, of the UDI Core Specification and Chapter 28, *"Metalanguage-to-Environment Interface"*, of the UDI Core Specification.

5.2 Binding/Unbinding Operations

These operations are used during the binding and unbinding of a device driver to a bridge driver.

NAME

udi_bus_device_ops_t *Device driver entry point ops vector*

SYNOPSIS

```
#include <udi.h>
#include <udi_physio.h>

typedef const struct {
     udi_channel_event_ind_op_t *channel_event_ind_op;
     udi_bus_bind_ack_op_t *bus_bind_ack_op;
     udi_bus_unbind_ack_op_t *bus_unbind_ack_op;
     udi_intr_attach_ack_op_t *intr_attach_ack_op;
     udi_intr_detach_ack_op_t *intr_detach_ack_op;
} udi_bus_device_ops_t;

/* Bus Device Ops Vector Number */
#define UDI_BUS_DEVICE_OPS_NUM           1
```

DESCRIPTION

A driver using the Physical I/O services will specify the `udi_bus_device_ops_t` as part of its `udi_init_info` in order to register its entry points for managing bus bindings and handling interrupts (as opposed to dispatching interrupts).

REFERENCES

`udi_init_info`, `udi_ops_init_t`, `udi_bus_bridge_ops_t`

EXAMPLE

The driver's initialization structure definitions might include the following:

```
#define MY_DEVICE_OPS 10 /* Ops for my child role */
#define MY_BRIDGE_OPS 11 /* Ops for my bridge role */
#define MY_BUS_META    1 /* Meta index for Bus Bridge Metalanguage */
static
   udi_bus_device_ops_t ddd_bus_device_ops = {
     ddd_bus_device_channel_event_ind,
     ddd_bus_bind_ack,
     ddd_bus_unbind_ack,
     ddd_intr_attach_ack,
     ddd_intr_detach_ack
};
...
static udi_ops_init_t ddd_ops_init_list[] = {
     {    MY_DEVICE_OPS,
          MY_BUS_META,
          UDI_BUS_DEVICE_OPS_NUM,
          0, /* chan_context_size */
          (udi_ops_vector_t *)&ddd_bus_device_ops },
     { 0 }
};
```

NAME

udi_bus_bridge_ops_t *Bridge driver entry point ops vector*

SYNOPSIS

```
#include <udi.h>
#include <udi_physio.h>

typedef const struct {
     udi_channel_event_ind_op_t *channel_event_ind_op;
     udi_bus_bind_req_op_t *bus_bind_req_op;
     udi_bus_unbind_req_op_t *bus_unbind_req_op;
     udi_intr_attach_req_op_t *intr_attach_req_op;
     udi_intr_detach_req_op_t *intr_detach_req_op;
} udi_bus_bridge_ops_t;

/* Bus Bridge Ops Vector Number */
#define UDI_BUS_BRIDGE_OPS_NUM           2
```

DESCRIPTION

A driver using the Physical I/O services and which acts as an “interrupt dispatcher” (as opposed to an “interrupt handler”) will specify the `udi_bus_bridge_ops_t` structure as part of its `udi_init_info` to register its entry points for managing bus bindings and interrupts attachments/detachments.

REFERENCES

`udi_init_info`, `udi_ops_init_t`, `udi_bus_device_ops_t`

EXAMPLE

The driver’s initialization structure definitions might include the following:

```
#define MY_DEVICE_OPS 10 /* Ops for my device role */
#define MY_BRIDGE_OPS 11 /* Ops for my bridge role */
#define MY_BUS_META   1  /* Meta index for Bus Bridge Metalanguage */
static
   udi_bus_bridge_ops_t ddd_bus_bridge_ops = {
     ddd_bus_bridge_channel_event_ind,
     ddd_bus_bind_req,
     ddd_bus_unbind_req,
     ddd_intr_attach_req,
     ddd_intr_detach_req
};
...
static udi_ops_init_t ddd_ops_init_list[] = {
     {    MY_DEVICE_OPS,
          MY_BUS_META,
          UDI_BUS_DEVICE_OPS_NUM,
          0, /* chan_context_size */
          (udi_ops_vector_t *)&ddd_bus_device_ops },
     {    MY_BRIDGE_OPS,
          MY_BUS_META,
          UDI_BUS_BRIDGE_OPS_NUM,
          0, /* chan_context_size */
          (udi_ops_vector_t *)&ddd_bus_bridge_ops },
     ...
};
```

NAME

udi_bus_bind_cb_t *Control block for bus bridge binding operations*

SYNOPSIS

```
#include <udi.h>
#include <udi_physio.h>

typedef struct {
     udi_cb_t gcb;
} udi_bus_bind_cb_t;

/* Bus Bind Control Block Group Number */
#define UDI_BUS_BIND_CB_NUM             1
```

MEMBERS

gcb is a generic control block header, which includes a pointer to the scratch space associated with this control block. The driver may use the scratch space while it owns the control block, but the values are not guaranteed to persist across channel operations.

DESCRIPTION

The bus bind control block is used between the bus device driver and the bus bridge driver to complete binding and unbinding over the bind channel.

This control block must be declared by specifying the control block index value `UDI_BUS_BIND_CB_NUM` in a `udi_cb_init_t` in the driver's `udi_init_info`.

The bus device driver obtains the `udi_bus_bind_cb_t` structure to use with the `udi_bus_bind_req` or `udi_bus_unbind_req` operation by calling `udi_cb_alloc` with a ***cb_idx*** that has been defined for the `UDI_BUS_BIND_CB_NUM` control block.

REFERENCES

`udi_bus_bind_cb_t`, `udi_cb_alloc`, `udi_bus_bind_req`, `udi_bus_bind_ack`, `udi_bus_unbind_req`, `udi_bus_unbind_ack`

NAME

udi_bus_bind_req *Request a binding to a bridge driver*

SYNOPSIS

```
#include <udi.h>
#include <udi_physio.h>

void udi_bus_bind_req (
     udi_bus_bind_cb_t *cb );
```

ARGUMENTS

cb	is a pointer to a bus bind control block.

TARGET CHANNEL

The target channel for this operation is the bind channel connecting a device driver with its parent bus bridge driver.

DESCRIPTION

A device driver uses this operation to bind to its parent bus bridge driver.

The device driver must prepare for the `udi_bus_bind_req` operation by allocating a bus bind control block (calling `udi_cb_alloc` with a `cb_idx` corresponding to the `UDI_BUS_BIND_CB_NUM` control block type).

Next, the device driver sends the bus bind control block to the bridge driver with a `udi_bus_bind_req` operation.

REFERENCES

```
udi_bus_bind_cb_t, udi_bus_bind_ack,
         udi_bus_unbind_req
```

NAME

udi_bus_bind_ack *Acknowledge a bus bridge binding*

SYNOPSIS

```
#include <udi.h>
#include <udi_physio.h>

void udi_bus_bind_ack (
     udi_bus_bind_cb_t *cb,
     udi_dma_constraints_t dma_constraints,
     udi_ubit8_t preferred_endianness,
     udi_status_t status );

/* Values for preferred_endianness */
#define UDI_DMA_BIG_ENDIAN               (1U<<5)
#define UDI_DMA_LITTLE_ENDIAN            (1U<<6)
#define UDI_DMA_ANY_ENDIAN               (1U<<0)
```

ARGUMENTS

cb is a pointer to a bus bind control block.

dma_constraints specifies the DMA constraints requirements of the bus bridge. The child driver must apply its own specific constraints attributes to this constraints object (using `udi_dma_constraints_attr_set`) before using it for its own DMA mappings.

preferred_endianness indicates the device endianness which works most effectively with the bridges in this path. It may be set to one of the following values:

UDI_DMA_LITTLE_ENDIAN
UDI_DMA_BIG_ENDIAN
UDI_DMA_ANY_ENDIAN

status indicates whether or not the binding was successful.

TARGET CHANNEL

The target channel for this operation is the bind channel connecting a bus bridge driver with one of its child device drivers.

DESCRIPTION

The `udi_bus_bind_ack` operation is used by a bridge driver to acknowledge binding with a child device driver (or failure to do so, as indicated by status), as requested by a `udi_bus_bind_req` operation. When a bind is acknowledged with this operation, the bridge driver must be prepared for DMA, PIO, or interrupt registration operations to be performed to the associated device and for the device to begin generating interrupts.

Some devices are bi-endian; that is, they can be placed in either a little-endian mode or a big-endian mode. ***preferred_endianness*** provides a hint to drivers for such devices, as to which endianness is likely to be most efficient. If this is set to UDI_DMA_ANY_ENDIAN, at least one interposed bridge is bi-endian, so either endianness can be supported without significant additional cost (i.e. without software byte swapping).

Drivers for fixed-endianness devices can ignore ***preferred_endianness***.

STATUS VALUES

UDI_STAT_CANNOT_BIND

WARNINGS

The control block must be the same control block as passed to the driver in the corresponding `udi_bus_bind_req` operation.

REFERENCES

`udi_bus_bind_cb_t, udi_bus_bind_req,`
`udi_channel_close`

NAME

udi_bus_unbind_req *Request a bridge driver unbinding (child to bridge)*

SYNOPSIS

```
#include <udi.h>
#include <udi_physio.h>

void udi_bus_unbind_req (
     udi_bus_bind_cb_t *cb );
```

ARGUMENTS

cb	is a pointer to a bus bind control block.

TARGET CHANNEL

The target channel for this operation is the bind channel connecting a device driver with its parent bus bridge driver.

DESCRIPTION

A device driver uses this operation to unbind from its parent bus bridge driver.

The device driver must prepare for the `udi_bus_unbind_req` operation by allocating a bus bind control block (calling `udi_cb_alloc` with a ***cb_idx*** that indicates a `udi_bus_bind_cb_t`).

Next, the device driver sends the bus unbind control block to the bridge driver with a `udi_bus_unbind_req` operation.

REFERENCES

`udi_bus_bind_cb_t, udi_bus_unbind_ack`

NAME

udi_bus_unbind_ack *Acknowledge a bus bridge unbinding*

SYNOPSIS

```
#include <udi.h>
#include <udi_physio.h>

void udi_bus_unbind_ack (
     udi_bus_bind_cb_t *cb );
```

ARGUMENTS

cb is a pointer to a bus bind control block.

TARGET CHANNEL

The target channel for this operation is the bind channel connecting a device driver with its parent bus bridge driver.

DESCRIPTION

The `udi_bus_unbind_ack` operation is used by a bridge driver to acknowledge an unbinding with a child device driver as requested by a `udi_bus_unbind_req` operation.

There is no status parameter associated with this operation; the bridge driver is expected to always be able to handle the unbind request and respond appropriately. If, for example, the bridge driver were to receive an unbind from a child without having first received a bind (or two unbinds in a row from the child), the bridge driver may log this condition but must always respond with this acknowledgment.

WARNINGS

The control block must be the same control block as passed to the driver in the corresponding `udi_bus_unbind_req` operation.

REFERENCES

`udi_bus_bind_cb_t`, `udi_bus_unbind_req`, `udi_channel_close`

5.3 Interrupt Registration Operations

These operations are used to register and de-register interrupt handlers for interrupt sources routed through the bus bridge.

NAME

udi_intr_attach_cb_t *Control block for interrupt registration operations*

SYNOPSIS

```
#include <udi.h>
#include <udi_physio.h>

typedef struct {
     udi_cb_t gcb;
     udi_index_t interrupt_idx;
     udi_ubit8_t min_event_pend;
     udi_pio_handle_t preprocessing_handle;
} udi_intr_attach_cb_t;

/* Bridge Attach Control Block Group Number */
#define UDI_BUS_INTR_ATTACH_CB_NUM       2
```

MEMBERS

gcb is a generic control block header, which includes a pointer to the scratch space associated with this control block. The driver may use the scratch space while it owns the control block, but the values are not guaranteed to persist across channel operations.

interrupt_idx is used to select one of possibly several interrupt sources from the interrupt handler's device that are managed by a particular interrupt dispatcher. Zero indicates the first interrupt source for the device, one indicates the second source, etc. The dispatcher driver will have previously associated the handler's device properties with its channel context during the initial binding sequence.

The definition of ***interrupt_idx*** is bus and device dependent, and must be determined by the driver via the *bus_type* instance attribute (see Chapter 15, *"Instance Attribute Management"* in the UDI Core Specification) in conjunction with the corresponding UDI bus bindings (see "Section 3: Bus Bindings"). For example, if *bus_type* is "`pci`" then the settings would be based on the definitions in the *UDI PCI Bus Binding Specification*.

min_event_pend is the minimum number of interrupt event control blocks that must be supplied to the dispatcher by the handler (via `udi_intr_event_rdy` operations) before the dispatcher will invoke the ***preprocessing_handle*** trans list at start label 0 to enable interrupts from the device. Once invoked at label 0, all subsequent PIO preprocessing will be invoked at label 1 until only a single interrupt event control block remains, at which point label 2 will be used to invoke PIO preprocessing. If no interrupt event control blocks remain, label 3 will be used whenever an interrupt causes the PIO preprocessing to be invoked. Interrupt processing will not exit the "label 3" state until at least ***min_event_pend*** control blocks are supplied, at which time label 0 will be invoked and the cycle will repeat. If the number of control blocks available falls below

min_event_pend, no action will be taken (PIO processing will continue to be invoked at label 1) until only one control block remains; if the number of control blocks moves back above ***min_event_pend*** without ever reaching the lower limit of 1 control block, no specific action will be taken and PIO processing will continue to be invoked at label 1 as normal.

The value for ***min_event_pend*** must be at least 2.

max_event_pend is the maximum number of interrupt events the device and driver can have pending at one time. This is a hint to the dispatcher, so it can determine how many event control blocks to allocate for maximum concurrency.

preprocessing_handle is an optional PIO handle to be used to preprocess interrupts. If not null, the interrupt dispatcher will execute the associated PIO transaction list before delivering the interrupt to the interrupt handler. If null, the handler must run in an interrupt region.

DESCRIPTION

The interrupt attach control block is used between the interrupt handler and the interrupt dispatcher to attach interrupt handler channels (i.e., register an interrupt handler for a particular interrupt source).

This control block must be declared by specifying the control block index value UDI_BUS_INTR_ATTACH_CB_NUM in a udi_cb_init_t in the driver's udi_init_info.

The bus device driver obtains the udi_intr_attach_cb_t structure to use with the udi_intr_attach_req by calling udi_cb_alloc with a ***cb_idx*** that has been associated with UDI_BUS_INTR_ATTACH_CB_NUM.

REFERENCES

udi_intr_attach_req, udi_intr_attach_ack, udi_init_info, udi_cb_init_t, udi_cb_alloc

NAME

udi_intr_attach_req *Request an interrupt attachment*

SYNOPSIS

```
#include <udi.h>
#include <udi_physio.h>

void udi_intr_attach_req (
    udi_intr_attach_cb_t *intr_attach_cb );
```

ARGUMENTS

intr_attach_cb is a pointer to an interrupt attach control block.

TARGET CHANNEL

The target channel for this operation is the bound bus bridge channel connecting a device driver with its parent bus bridge driver, established during the initial binding between the child device and its bus bridge.

DESCRIPTION

An interrupt handler driver uses this operation to request an interrupt dispatcher driver to begin accepting interrupts for the handler driver's device and delivering them to the handler. If a device has multiple interrupt sources, the interrupt handler indicates which one to attach by using ***interrupt_idx*** in the control block.

The handler driver must prepare for the `udi_intr_attach_req` operation by allocating an interrupt attach control block, filling in the relevant members of the control block, and spawning its end of a new interrupt event channel.

The new channel will be used to send interrupt event operations from the interrupt dispatcher to the handler. The handler driver spawns its end by calling `udi_channel_spawn`, passing in its end of the bus bridge channel and setting spawn index equal to the ***interrupt_idx*** for this attachment. Using ***interrupt_idx*** for the spawn index ensures that it is unique with respect to any other spawn operations in progress on the same bus bridge channel, allowing the environment use this index to match up the two parts of the spawn operation.

Next, the interrupt handler sends the interrupt control block to the dispatcher driver with a `udi_intr_attach_req` operation.

When the interrupt dispatcher driver receives this request, it spawns its end of the new interrupt event channel, calling `udi_channel_spawn`, passing in its end of the bus bridge channel and using ***interrupt_idx*** for the spawn index. The dispatcher then sends a `udi_intr_attach_ack` operation back on the original channel to the handler driver, which may now reuse the spawn index.

The handler driver can issue an `udi_intr_attach_req` for an ***interrupt_idx*** that is already attached (i.e., uses the same ***interrupt_idx*** as an existing attachment), in which case the `udi_intr_attach_req` simply changes the interrupt pre-processing for that interrupt source and ignores all other parameters. No channels are spawned and no other actions are taken when interrupt pre-processing is updated through this re-attachment operation.

If ***preprocessing_handle*** is not null (use `UDI_HANDLE_IS_NULL` to test it), events on this channel will be preprocessed. Otherwise, the handler driver must ensure that the handler's end of the new channel is anchored in an interrupt region (i.e. a region with the "interrupt" attribute set in the driver's static driver properties; see Section 1.4, "Extensions to Static Driver Properties," on page 1-2).

Some of the advantages of interrupt preprocessing are that the interrupt handler does not need to be in an interrupt region and that the `udi_intr_event_ind` handler code is not restricted in the operations it may perform before responding with `udi_intr_event_rdy`, unlike in the non-preprocessed case. There are, however, some restrictions on interrupt preprocessing as discussed in detail in the `udi_intr_event_ind` operation description.

When the preprocessing trans list is executed, the dispatcher will invoke the trans list at one of four ***start_label*** entry points (i.e. the trans list should contain `UDI_PIO_LABEL` transactions for each of these values, with the exception of zero which implies the beginning of the transaction list):

0 The dispatcher will invoke the preprocessing trans list with this ***start_label*** value to enable or re-enable interrupts (at the device level), once the dispatcher has received at least ***min_event_pend*** interrupt control blocks. When started from this point, the trans list is responsible for enabling interrupts from the device then proceeding as for ***start_label*** 1 below (if applicable). Drivers that cannot determine whether their device asserted an interrupt must instead exit with a `UDI_INTR_UNCLAIMED` result code and not perform label 1 processing.

1 The dispatcher will invoke the preprocessing trans list at this label to handle the normal interrupt condition. The trans list should process the indicated interrupt(s), placing any interrupt information in the associated buffer, and return an appropriate exit code to the dispatcher as defined on page 5-27.

2 The dispatcher will invoke the preprocessing trans list at this label to handle the interrupt overrun condition. This condition occurs when an unmasked interrupt occurs and the dispatcher has only one `udi_intr_event_cb_t` available from the handler; any subsequent interrupts would not have an event control block or associated buffer to handle the interrupt. The trans list must check for an actual interrupt: if the device is not interrupting, the trans list must return `UDI_INTR_UNCLAIMED`; if the trans list handles the interrupt completely and no handler notification is needed, then `UDI_INTR_NO_EVENT` must be returned; in either of these cases the overrun situation is not encountered. Otherwise, the trans list must process the interrupt and then disable further interrupts from the device (if possible). Interrupts will be re-enabled by an entry into the trans list at

start_label value 0 after ***min_event_pending*** number of additional udi_intr_event_cb_t control blocks have been supplied to the dispatcher by the handler.

3 The dispatcher will invoke the preprocessing trans list at this label to handle any interrupts received during the overrun condition. This entry point is used if interrupts are received after issuing a label 2 preprocessing operation to disable interrupts (i.e. the label position 2 entry point is unable to disable interrupts from the device and/or if the device is used in a shared interrupt situation) or if any interrupts are received before the dispatcher has enough interrupt event control blocks to use label 1. This trans list must determine if additional interrupts are being signalled by the device and, if so, dismiss the interrupt while discarding any associated data and return a status of UDI_INTR_NO_EVENT. If the device is not indicating an interrupt, UDI_INTR_UNCLAIMED must be returned by the trans list. When the transaction list is entered at this label, the exit code must be one of UDI_INTR_NO_EVENT or UDI_INTR_UNCLAIMED. All PIO preprocessing trans lists entered at label 3 should operate as if there is no associated control block or buffer, even if the handler has supplied more interrupt control blocks; interrupt processing will resume normally once the handler has supplied the ***min_event_pend*** number of control blocks and the PIO trans list has been entered at label 0.

Any interrupts for the selected interrupt source that are processed by the dispatcher after receiving a udi_intr_attach_req with a non-null ***preprocessing_handle*** will be preprocessed. This means that the dispatcher will use udi_pio_trans to execute the specified trans list (associated with ***preprocessing_handle***). This trans list must include all operations necessary to de-assert its interrupt (if it was asserted) in all cases.

When the dispatcher calls udi_pio_trans it will set the ***buf*** argument to the buffer passed in the udi_intr_event_cb_t from the udi_intr_event_rdy. This buffer must have been pre-allocated by the handler to contain the data to be returned by the interrupt dispatcher; the buffer will not be extended as part of the interrupt handling operation. The trans list can use this buffer to pass data to the handler driver's upper-level handler.

If the bus type supports event info, the dispatcher driver must set the ***mem_ptr*** argument to point to a memory block containing the event info for this interrupt. Otherwise, ***mem_ptr*** must be set to NULL.

The preprocessing trans list must only use the first byte of the control block's scratch space to pass back the status of the interrupt preprocessing as specified in the udi_intr_event_cb_t description. No other references can be made to scratch space since the size and contents are unspecified beyond the first byte.

The ***result*** from the `udi_pio_trans` call (set with UDI_PIO_END) is used to determine the disposition of the interrupt, as described in **udi_intr_event_cb_t** on page 5-27 and **udi_intr_event_ind** on page 5-29.

The bridge driver must free the ***preprocessing_handle*** after it has finished with it (as a result of either another `udi_intr_attach_req` with a new handle or a `udi_intr_detach_req`), by using `udi_pio_unmap`.

WARNINGS The ***mem_ptr*** value must follow the rules for memory object usage described in Section 5.2.1.1, "Using Memory Pointers with Asynchronous Service Calls," on page 5-2 of the UDI Core Specification.

REFERENCES `udi_intr_attach_cb_t, udi_channel_spawn, udi_intr_attach_ack, udi_intr_event_ind, udi_pio_trans, udi_pio_map, udi_pio_unmap`

NAME

udi_intr_attach_ack *Acknowledge an interrupt attachment*

SYNOPSIS

```
#include <udi.h>
#include <udi_physio.h>

void udi_intr_attach_ack (
     udi_intr_attach_cb_t *intr_attach_cb,
     udi_status_t status );
```

ARGUMENTS

intr_attach_cb is a pointer to an interrupt attach control block.

status indicates whether or not the attachment was successful.

TARGET CHANNEL

The target channel for this operation is the bound bus bridge channel connecting a bus bridge driver with one of its child device drivers, established during the initial binding between the child device and its bus bridge.

PROXIES

udi_intr_attach_ack_unused *Proxy for udi_intr_attach_ack*

`udi_intr_attach_ack_op_t` **`udi_intr_attach_ack_unused`**;

`udi_intr_attach_ack_unused` may be used as a device driver's `udi_intr_attach_ack` entry point if the driver never calls `udi_intr_attach_req` (and therefore expects to receive no acknowledgements).

DESCRIPTION

The `udi_intr_attach_ack` operation is used by an interrupt dispatcher driver to acknowledge attachment of an interrupt handler (or failure to do so, as indicated by status), as requested by a `udi_intr_attach_req` operation.

Before sending the acknowledgment, the interrupt dispatcher driver must spawn and anchor its end of the new interrupt event channel, setting the spawn index equal to the ***interrupt_idx*** for this attachment.

interrupt_idx in the returned control block must be the same as that passed to `udi_intr_attach_req`.

Upon failure indication, the handler driver must be sure to close the half-spawned channel, by calling `udi_channel_close`.

STATUS VALUES

UDI_OK- The handler was successfully registered for the indicated interrupts and normal interrupt processing will occur. The ***preprocessing_handle*** argument returned in the control block must be UDI_NULL_PIO_HANDLE.

UDI_STAT_MISTAKEN_IDENTITY – This device has no interrupt source corresponding to the ***interrupt_idx*** value in the control block. The ***preprocessing_handle*** member of the control block is unchanged and the corresponding PIO handle ownership is returned to the interrupt handler.

WARNINGS The control block must be the same control block as passed to the driver in the corresponding `udi_intr_attach_req` operation.

REFERENCES `udi_intr_attach_cb_t`, `udi_intr_attach_req`, `udi_intr_detach_req`, `udi_channel_close`

NAME

udi_intr_detach_cb_t *Control block for interrupt detachment operations*

SYNOPSIS

```
#include <udi.h>
#include <udi_physio.h>

typedef struct {
     udi_cb_t gcb;
     udi_index_t interrupt_idx;
} udi_intr_detach_cb_t;

/* Bridge Detach Control Block Group Number */
#define UDI_BUS_INTR_DETACH_CB_NUM        3
```

MEMBERS

gcb is a generic control block header, which includes a pointer to the scratch space associated with this control block. The driver may use the scratch space while it owns the control block, but the values are not guaranteed to persist across channel operations.

interrupt_idx is used to select one of possibly several interrupt sources from the interrupt handler's device that are managed by a particular interrupt dispatcher. Zero indicates the first interrupt source for the device, one indicates the second source, etc. The dispatcher driver will have previously associated the handler's device properties with its channel context during the initial binding sequence.

DESCRIPTION

The interrupt detach control block is used between the interrupt handler and the interrupt dispatcher to detach interrupt handler channels (i.e., deregister an interrupt handler for a particular interrupt source).

This control block must be declared by specifying the control block index value `UDI_BUS_INTR_DETACH_CB_NUM` in a `udi_cb_init_t` in the driver's `udi_init_info`.

The bus device driver obtains the `udi_intr_detach_cb_t` structure to use with the `udi_intr_detach_req` by calling `udi_cb_alloc` with a ***cb_idx*** that has been associated with `UDI_BUS_INTR_DETACH_CB_NUM`.

REFERENCES

`udi_intr_detach_req, udi_intr_detach_ack, udi_init_info, udi_cb_init_t, udi_cb_alloc`

NAME **udi_intr_detach_req** *Request an interrupt detachment*

SYNOPSIS

```
#include <udi.h>
#include <udi_physio.h>

void udi_intr_detach_req (
     udi_intr_detach_cb_t *intr_detach_cb );
```

ARGUMENTS ***intr_detach_cb*** is a pointer to an interrupt detach control block.

TARGET CHANNEL The target channel for this operation is the bound bus bridge channel connecting a device driver with its parent bus bridge driver, established during the initial binding between the child device and its bus bridge.

DESCRIPTION When the interrupt handler driver wishes to detach its handler from the interrupt dispatcher, it invokes the `udi_intr_detach_req` channel operation. A prior attachment must have been completed, as indicated to the interrupt handler driver by receipt of the attach acknowledgment.

interrupt_idx in the control block must be the same as that passed to `udi_intr_attach_req` (and returned with `udi_intr_attach_ack`).

REFERENCES `udi_intr_detach_cb_t, udi_intr_detach_ack`

NAME

udi_intr_detach_ack *Acknowledge an interrupt detachment*

SYNOPSIS

```
#include <udi.h>
#include <udi_physio.h>

void udi_intr_detach_ack (
     udi_intr_detach_cb_t *intr_detach_cb );
```

ARGUMENTS

intr_detach_cb is a pointer to an interrupt detach control block.

TARGET CHANNEL

The target channel for this operation is the bound bus bridge channel connecting a bus bridge driver with one of its child device drivers, established during the initial binding between the child device and its bus bridge.

PROXIES

udi_intr_detach_ack_unused *Proxy for udi_intr_detach_ack*

```
udi_intr_detach_ack_op_t udi_intr_detach_ack_unused;
```

udi_intr_detach_ack_unused may be used as a device driver's udi_intr_detach_ack entry point if the driver never calls udi_intr_detach_req (and therefore expects to receive no acknowledgements).

DESCRIPTION

After detaching the interrupt handler and closing the local end of the corresponding interrupt event channel, the interrupt dispatcher driver sends the control block back to the interrupt handler using the channel operation, udi_intr_detach_ack.

The dispatcher should discard any udi_intr_event_cb_t control blocks currently held via udi_cb_free before acknowledging the detach operation. Additionally, any interrupt event control blocks received while not attached should be discarded in a similar manner.

The ***interrupt_idx*** in the returned control block must be the same as that passed to udi_intr_detach_req.

Upon receipt of the udi_intr_detach_ack, the interrupt handler must close its end of the interrupt event channel (if it did not already do so as a result of a udi_channel_event_ind).

The interrupt handler may free the control block using udi_cb_free or reuse it for another detachment.

WARNINGS

The control block must be the same control block as passed to the driver in the corresponding udi_intr_detach_req operation.

REFERENCES

udi_intr_detach_cb_t, udi_intr_detach_req, udi_cb_free

5.4 Interrupt Event Operations

These operations are used to deliver and respond to interrupt events between interrupt dispatchers and interrupt handlers.

Interrupt events may be handled in one of two ways: with interrupt preprocessing, or by using interrupt regions. See **udi_intr_attach_req** on page 5-15 for details on interrupt preprocessing. Any region using `udi_intr_event_ind` or `udi_intr_event_rdy` without interrupt preprocessing enabled must be an *interrupt region*; i.e. it must have its `type` region attribute set to `interrupt` (see Table 1-1, “Physical I/O Region Attributes,” on page 1-3). The primary region of a driver instance cannot be an interrupt region.

All environment services are available to interrupt handlers, but since interrupt regions can require critical resources to be held while executing, drivers should minimize the amount of time spent executing in interrupt regions.

There is, however, a restriction on the *order* in which service calls and channel operations can be invoked from an interrupt handler, when interrupt preprocessing is not used. For details, see `udi_intr_event_ind` and `udi_intr_event_rdy`.

NAME

udi_intr_handler_ops_t *Interrupt handler ops vector*

SYNOPSIS

```
#include <udi.h>
#include <udi_physio.h>

typedef const struct {
     udi_channel_event_ind_op_t *channel_event_ind_op;
     udi_intr_event_ind_op_t *intr_event_ind_op;
} udi_intr_handler_ops_t;

/* Interrupt Handler Ops Vector Number */
#define UDI_BUS_INTR_HANDLER_OPS_NUM      3
```

DESCRIPTION

A driver which wishes to register for handling interrupts (as opposed to "dispatching" interrupts declares the `udi_intr_handler_ops_t` structure to define its entry point for receiving interrupt events.

REFERENCES

`udi_init_info`, `udi_ops_init_t`, `udi_intr_dispatcher_ops_t`

EXAMPLE

The driver's initialization structure definitions might include the following:

```
#define MY_INTR_HANDLER_OPS 2 /* My interrupt handler ops */
#define MY_BUS_META 1 /* Meta index for the Bus Bridge Metalanguage */
static udi_intr_handler_ops_t
     ddd_intr_handler_ops = {
          ddd_intr_handler_channel_event_ind,
          ddd_intr_event_ind
     };
...
static udi_ops_init_t ddd_ops_init_list[] = {
     {    MY_INTR_HANDLER_OPS,
          MY_BUS_META,
          UDI_BUS_INTR_HANDLER_OPS_NUM,
          0, /* chan_context_size */
          (udi_ops_vector_t *)&ddd_intr_handler_ops },
     { 0 }
};
```

NAME

udi_intr_dispatcher_ops_t *Interrupt dispatcher ops vector*

SYNOPSIS

```
#include <udi.h>
#include <udi_physio.h>

typedef const struct {
     udi_channel_event_ind_op_t *channel_event_ind_op;
     udi_intr_event_rdy_op_t *intr_event_rdy_op;
} udi_intr_dispatcher_ops_t;

/* Interrupt Dispatcher Ops Vector Number */
#define UDI_BUS_INTR_DISPATCH_OPS_NUM     4
```

DESCRIPTION

A bus driver which delivers interrupt indications uses the `udi_intr_dispatcher_ops_t` to declare the interface operations for receiving interrupt acknowledgements from the interrupt handler.

REFERENCES

`udi_init_info`, `udi_ops_init_t`, `udi_intr_handler_ops_t`

EXAMPLE

The driver's initialization structure definitions might include the following:

```
#define MY_INTR_DISP_OPS 2 /* My interrupt dispatcher ops */
#define MY_BUS_META 1 /* Meta index for the Bus Bridge Metalanguage */
static udi_intr_dispatcher_ops_t
     ddd_intr_dispatcher_ops = {
          ddd_intr_channel_event_ind,
          ddd_intr_event_rdy
     };
...
static udi_ops_init_t ddd_ops_init_list[] = {
     {    MY_INTR_DISP_OPS,
          MY_BUS_META,
          UDI_BUS_INTR_DISPATCH_OPS_NUM,
          0, /* chan_context_size */
          (udi_ops_vector_t *)&ddd_intr_dispatcher_ops
     },
     { 0 }
};
```

NAME

udi_intr_event_cb_t *Control block for interrupt event ops*

SYNOPSIS

```
#include <udi.h>
#include <udi_physio.h>

typedef struct {
     udi_cb_t gcb;
     udi_buf_t *event_buf;
     udi_ubit16_t intr_result;
} udi_intr_event_cb_t;

/* Flag values for interrupt handling */
#define UDI_INTR_UNCLAIMED                (1U<<0)
#define UDI_INTR_NO_EVENT                 (1U<<1)

/* Bus Interrupt Event Control Block Group Number */
#define UDI_BUS_INTR_EVENT_CB_NUM         4
```

MEMBERS

gcb is a generic control block header, which includes a pointer to the scratch space associated with this control block. The driver may use the scratch space while it owns the control block, but the values are not guaranteed to persist across channel operations.

event_buf This buffer (if any) must be preallocated by the handler before being passed to the dispatcher and must contain enough valid bytes to handle the largest PIO trans list requirements; the dispatcher's PIO trans operations cannot extend the size of this buffer.

intr_result is, in the non-preprocessing case, set by the interrupt handler before invoking `udi_intr_event_rdy`, to indicate whether or not its device asserted the interrupt. In the preprocessing case, ***intr_result*** is set by the dispatcher to the ***result*** value from the `udi_pio_trans` call used to preprocess the interrupt, and passed to the handler via the `udi_intr_event_ind` channel operation.

`UDI_INTR_UNCLAIMED` is set by the non-preprocessing handler in the ***intr_result*** field for the `udi_intr_event_rdy` operation to indicate that the corresponding device did not generate the interrupt and that the dispatcher should proceed to process any other shared interrupts for this interrupt line. In the preprocessing case, this status is indicated by the preprocessing PIO trans list by setting this bit in the first byte of the control block's scratch space; the dispatcher will not pass the interrupt event indication to the handler and will check other devices as in the non-preprocessing case.

`UDI_INTR_NO_EVENT` is not used for interrupts occurring in the non-preprocessing case, but may be set in the first byte of the control block's scratch space by the PIO trans list in the preprocessing case to indicate that the PIO trans list handled the interrupt in its entirety and that no interrupt event should be delivered to the handler.

The `UDI_INTR_NO_EVENT` value may be set for newly allocated interrupt event control blocks allocated by the handler and passed to the dispatcher via the `udi_intr_event_rdy` operation.

The ***`intr_result`*** value returned in the `udi_intr_event_rdy` call for the preprocessing case is ignored by the dispatcher.

The dispatcher will zero the first byte of the control block scratch space before initiating the `udi_pio_trans` on the preprocessing PIO trans list in the preprocessing case; the PIO trans list does not need to modify the byte unless specific bits must be set. The remaining bytes of scratch space are unspecified and must not be used by the preprocessing trans list.

Drivers that cannot tell whether or not their device actually asserted an interrupt, or whose device's interrupts are for some other reason non-sharable, must include a "nonsharable_interrupt" declaration in their static driver properties file (see Section 1.4, "Extensions to Static Driver Properties," on page 1-2), and must not use `UDI_INTR_UNCLAIMED`.

DESCRIPTION

The interrupt event control block is used between the interrupt handler and the interrupt dispatcher to deliver and acknowledge interrupt events.

If the interrupt event was pre-processed (as indicated by the initial value of ***`intr_result`*** passed to `udi_intr_event_ind`), the ***`event_buf`*** will contain any data filled in during the first-level handler by using the device driver's pre-registered PIO transaction list. (See ***`preprocessing_handle`*** in `udi_intr_attach_cb_t`.)

Otherwise, ***`event_buf`*** will contain bus-type specific event information, as defined in the bus binding specification for the type of bus to which the device is attached. If the size of the event information is greater than the valid buffer size, only the information that fits in the pre-validated buffer region will be returned. If the event information size is less than the valid buffer size, the remaining bytes will be part of the buffer's valid data range, but the contents are unspecified.

If a driver supports a device which might make use of interrupt event info, it must first determine what type of bus the device is plugged into, by looking at its driver instance attributes. It may then interpret the event info using the corresponding bus binding. See Chapter 6, *"Introduction to Bus Bindings"*, for information on the format of interrupt event info.

REFERENCES

`udi_intr_attach_cb_t`

NAME

udi_intr_event_ind *Interrupt event indication*

SYNOPSIS

```
#include <udi.h>
#include <udi_physio.h>

void udi_intr_event_ind (
     udi_intr_event_cb_t *intr_event_cb,
     udi_ubit8_t flags );

/* Values for flags */
#define UDI_INTR_MASKING_NOT_REQUIRED     (1U<<0)
#define UDI_INTR_OVERRUN_OCCURRED         (1U<<1)
#define UDI_INTR_PREPROCESSED             (1U<<2)
```

ARGUMENTS

intr_event_cb is a pointer to an interrupt event control block.

flags specifies optional flags, described below.

TARGET CHANNEL

The target channel for this operation is the interrupt event channel for this attachment, jointly spawned by the interrupt handler and interrupt dispatcher during interrupt attachment (see `udi_intr_attach_req`, `udi_intr_attach_ack`). The interrupt handler driver can use the channel context pointer for this channel to distinguish this event from events for other attached interrupts.

DESCRIPTION

Upon receipt of an interrupt condition that may have been generated by a particular interrupt source, an interrupt dispatcher will prepare to deliver the interrupt to the appropriate handler. If the handler has requested interrupt preprocessing, it will execute the pre-registered PIO trans list (see **udi_intr_attach_req** on page 5-15) before delivering the interrupt to the handler driver.

The dispatcher must not allocate interrupt event control blocks to deliver the interrupt to the handler; the handler must have already provided a control block to be used for this purpose. If no control blocks are currently available, the dispatcher should execute the preprocessing trans list (if specified) starting at label 3 (as described by `udi_intr_attach_req`) and then exit without attempting to notify the handler of the interrupt.

If the preprocessing transaction list handles the interrupt in its entirety such that there is no need for the dispatcher to signal the interrupt event to the handler, the `UDI_INTR_NO_EVENT` flag should be set in the first byte of the control block's scratch space (see page 4-22) and the dispatcher will simply exit without performing the `udi_intr_event_ind` operation to the handler.

Since some interrupt handlers change between non-preprocessing and preprocessing modes, the presence of the `UDI_INTR_PREPROCESSED` bit in ***flags*** indicates whether or not this particular interrupt was pre-processed. The dispatcher must set `UDI_INTR_PREPROCESSED` in ***flags*** if this interrupt was preprocessed; otherwise this bit will never be set.

The following applies only to the preprocessing case:

> The ***result*** value from the `udi_pio_trans` call used to preprocess the interrupt is used by the dispatcher to set ***intr_result*** in the interrupt event control block. If either the `UDI_INTR_UNCLAIMED` or the `UDI_INTR_NO_EVENT` flags were set in the first byte of scratch space, no `udi_intr_event_ind` operation will be sent to the handler driver; otherwise, the dispatcher must set ***event_buf*** to the buffer used with the `udi_pio_trans` call, set ***intr_result*** to the ***result*** value, and deliver the event to the dispatcher using `udi_intr_event_ind` with the `UDI_INTR_PREPROCESSED` flag.

The following applies only to the non-preprocessing case:

> The dispatcher driver must fill in the ***event_buf*** buffer with the bus type-specific event info for the interrupt, if any, and deliver the event to the appropriate interrupt handler using the channel operation, `udi_intr_event_ind`.

When the interrupt handler receives and interrupt event indication it must process that indication and either respond back to the dispatcher using `udi_intr_event_res` (after deasserting the device interrupt condition) or it must act as an intermediary dispatcher for child interrupt handlers. An intermediary dispatcher must execute any child interrupt handler's registered preprocessing PIO trans lists and/or issue `udi_intr_event_ind` operations to the child's interrupt event channel, passing the subsequent response back to the parent dispatcher.

If the interrupt was not preprocessed, the `udi_intr_event_ind` operation must execute in an interrupt region, and the driver must process the interrupt as described in the previous paragraph before performing any other channel operations or asynchronous service calls, with the exception of `udi_pio_trans`, and must not depend on any additional callbacks or channel ops entry points, except the `udi_pio_trans` callback(s), in order to complete the sequence needed to invoke the required interrupt event operation.

If the interrupt was preprocessed, there are no restrictions on the service calls and channel operations that `udi_intr_event_ind` may invoke before invoking `udi_intr_event_ind` or `udi_intr_event_rdy`, since the interrupt condition was already dismissed in the first-level handler.

The ***flags*** values are interpreted as follows:

`UDI_INTR_MASKING_NOT_REQUIRED` - this flag indicates that an interrupt handler that is also an interrupt dispatcher does not need to mask off the interrupt before passing the event on to the next level (because a higher level interrupt dispatcher interprets interrupts as one-shot events—rather than continuous assertion—and will not pass continuous assertion through).

`UDI_INTR_OVERRUN_OCCURRED` - this flag indicates that the ***`preprocessing_handle`*** PIO trans list was executed one or more times at start label 3 and did not return `UDI_INTR_UNCLAIMED`. This is an indication to the handler that one or more interrupts from the device were dismissed and their associated data was discarded prior to the current event being indicated.

`UDI_INTR_PREPROCESSED` - this flag indicates that the preprocessing PIO trans list was called for the associated interrupt. If clear, no trans list was executed and the handler must operate in the non-preprocessing mode. This will only occur if the device driver previously attached the interrupt without preprocessing and then invoked `udi_intr_attach_req` again with preprocessing enabled.

REFERENCES

`udi_intr_event_cb_t, udi_intr_attach_req, udi_intr_event_rdy`

NAME **udi_intr_event_rdy** *Acknowledge an interrupt event*

SYNOPSIS

```
#include <udi.h>
#include <udi_physio.h>

void udi_intr_event_rdy (
     udi_intr_event_cb_t *intr_event_cb );
```

ARGUMENTS ***intr_event_cb*** is a pointer to an interrupt event control block.

TARGET CHANNEL The target channel for this operation is the interrupt event channel for this attachment, jointly spawned by the interrupt handler and interrupt dispatcher during interrupt attachment (see `udi_intr_attach_req`, `udi_intr_attach_ack`). The interrupt dispatcher driver can use the channel context pointer for this channel to distinguish this response from responses for other attached interrupts.

DESCRIPTION An interrupt handler driver uses `udi_intr_event_rdy` to send an interrupt event control block to the dispatcher driver. These event control blocks allow the dispatcher to notify the handler of interrupt events, so the handler should endeavor to keep a number of these posted to the dispatcher to avoid missing interrupts. When the dispatcher indicates an interrupt, the handler should process the interrupt quickly and return the control block to the dispatcher to be used for future interrupt indications.

For a leaf interrupt source, this often means accessing some device-specific register which causes the device to stop asserting the interrupt. Leaf drivers must call `udi_intr_event_rdy` before returning from `udi_intr_event_ind`. Bridge drivers may complete the interrupt handling by passing the `udi_intr_event_rdy` on to their parent bridge, or may invoke another interrupt handler with `udi_intr_event_ind`.

For an interrupt handler which is also a dispatcher, `udi_intr_event_rdy` should be called immediately if the `UDI_INTR_MASKING_NOT_REQUIRED` flag is set, or after masking the interrupt if possible. If the interrupt could not be masked, then `udi_intr_event_rdy` should be called upon receipt of a `udi_intr_event_rdy` operation from the child handler driver.

In either case, if not using interrupt preprocessing, the handler driver must set ***intr_result*** in the control block to zero if its device asserted the interrupt (or the interrupt is non-sharable as indicated by the “nonsharable_interrupt” declaration) or to `UDI_INTR_UNCLAIMED` if the device was not the source of the interrupt. When using interrupt preprocessing, ***intr_result*** is ignored for `udi_intr_event_ind`.

The `udi_intr_event_rdy` operation must only be used when the handler is attached for that interrupt source (via `udi_intr_attach_req`).

The handler determines the number of interrupts indications that may be handled for this device by controlling the number of interrupt event control blocks that are used. The handler may allocate more interrupt control blocks at any time that it is attached to the interrupt dispatcher and provide those control blocks to the dispatcher for processing via the `udi_intr_event_rdy` operation; the ***`intr_result`*** field of these newly allocated control blocks should be set to `UDI_INTR_NO_EVENT`. If the handler wishes to reduce the number of interrupts being handled, it should reduce the number of interrupt control blocks available to the dispatcher by using `udi_channel_op_abort` and subsequently deallocating the aborted control blocks; the handler should not simply deallocate control blocks delivered via the `udi_intr_event_ind` operation since the interrupt must be acknowledged back to the dispatcher via the `udi_intr_event_rdy` operation.

WARNINGS The control block must be the same control block as passed to the driver in the corresponding `udi_intr_event_ind` operation.

REFERENCES `udi_intr_event_cb_t, udi_intr_attach_ack, udi_intr_event_ind, udi_channel_op_abort`

5.5 Static Properties Bindings

The driver category to be used with the “category” declaration by a portable implementation of the Bus Bridge Metalanguage Library shall be “Bus Bridges”.

5.6 Instance Attribute Bindings

One enumeration attribute is defined for all uses of the bus bridge metalanguage: `bus_type`. This attribute, of type `UDI_ATTR_STRING`, is set to the name of the I/O bus type to which the child adapter is connected, as defined by the relevant Bus Binding.

Additional enumeration attributes are specified in each Bus Binding to apply to each bus type.

5.7 Bus Bridge Trace Events

The Bus Bridge Metalanguage defines the use of `UDI_TREVENT_META_SPECIFIC_1` for bus bridge drivers when they are about to call `udi_pio_trans` to execute a preprocessing transaction list.

The `UDI_TREVENT_IO_SCHEDULED` and `UDI_TREVENT_IO_COMPLETED` events are not defined for use with the Bus Bridge Metalanguage.

5.8 Bus Bridge Metalanguage States

The following events change the state of a device driver with respect to the Bus Bridge Metalanguage:

1. Binding initiated
2. Binding complete
3. Interrupt attachment initiated
4. Interrupt attachment complete
5. Interrupt detachment initiated
6. Interrupt detachment complete
7. Unbinding initiated
8. Unbinding complete

The following events change the state of a bus bridge driver with respect to the Bus Bridge Metalanguage:

1. Binding to a new child
2. New interrupt attached
3. Interrupt detached
4. Child unbound

5.9 Bus Bridge Status Codes

No metalanguage-specific status codes are defined for the Bus Bridge Metalanguage.

UDI Physical I/O Specification

Section 3: Bus Bindings

Introduction to Bus Bindings 6

Some usage details of UDI Physical I/O services and metalanguages vary depending on the physical I/O bus used, requiring a Bus Binding definition to provide these details. This chapter defines the general requirements on Bus Binding definitions, and lists the items that must be specified in a UDI Bus Binding specification.

Note – There are no aspects of DMA services that can vary from one bus binding to another.

6.1 Normative References

Some bus bindings references non-UDI standards that, through reference in the bus binding specification, constitute provisions of the UDI bus binding. Each such bus binding shall list all normative references.

6.2 Header Files

Each bus binding shall define a bus binding header file that the driver must include to obtain bus binding declarations for a particular bus type. This header must be included by drivers that support devices on this bus type. It must be included after `udi.h` and `udi_physio.h`.

6.3 PIO Bindings

6.3.1 udi_pio_map

The definition of the ***regset_idx*** parameter to `udi_pio_map` is bus and device dependent. Each bus binding defines legal values for ***regset_idx*** and the semantics thereof. Drivers must set ***regset_idx*** to a value appropriate for the bus type of their supported device.

6.4 Interrupt Bindings

6.4.1 Interrupt Index Values

Interrupt index values for `udi_intr_attach_req` are bus-specific. Each bus binding must define the range of legal ***interrupt_idx*** values and their semantics.

6.4.2 Event Info

Interrupt event info is a bus-specific piece of extra information delivered from the interrupting device hardware along with the interrupt.

The size and structure of event info associated with an interrupt is bus type dependent. Each bus bindings defines the size and semantics of event info for their bus type.

6.5 Instance Attribute Bindings

A number of instance attributes are bus-specific in nature and are thus defined in bus bindings. These attributes shall include a set of enumeration attributes that are required to be set by any bridge driver (or mapper) when it enumerates the children attached to a bus. (See **udi_enumerate_ack** in the UDI Core Specification.)

6.5.1 Enumeration Attributes

In all cases, the “`bus_type`” attribute is considered an enumeration attribute, and the value used to identify each bus type must be defined in the corresponding bus binding. Each bus binding shall also specify additional enumeration attributes. Any enumerator for a particular bus type must support the specified enumeration attributes for the corresponding bus binding.

6.5.2 Filter Attributes

A subset of enumeration attributes must be specified as filter attributes. Filter attributes may be used with `udi_enumerate_req` to specify enumeration filters. Any enumerator for a particular bus type must support the specified filter attributes for the corresponding bus binding.

6.5.3 Generic Attributes

Each bus binding shall specify the form of string value for each of the generic attributes, “`identifier`”, “`address_locator`”, “`physical_locator`” and “`physical_slot`”, for the corresponding bus type. The locator attribute is used to distinguish one instance of the same type of device from another. Locator attribute values must be unique for all children of a particular parent bus bridge.

6.5.4 Parent-Visible Attributes

Some bus bindings may also specify parent-visible attributes. Any bus bridge driver for a particular bus type must support the specified parent-visible attributes for the corresponding bus binding.

UDI Physical I/O Specification

Section 4: Appendices

Glossary *A*

atomic transaction	an I/O bus transaction which is indivisible with respect to other transactions. Other concurrent transactions will either see the state of the referenced memory (or device register) as it was before the atomic transaction or after it, but not an intermediate value. Further, the adapter being accessed will see the access as exactly one transaction. For example, a WRITE4 transaction (4-byte write) is atomic if no observer of the same address can see a mixture of old and new byte values, and two WRITE4's for the same address are atomic with respect to each other if one overwrites the other without mixing byte values.
bridge	see ***bus bridge***.
bridge driver	see ***bus bridge driver***.
bridge metalanguage	see ***bus bridge metalanguage***.
bus address	see ***card-relative address***.
bus bridge	hardware which connects one kind of bus to another. For example, the bridge(s) between the PMI and primary I/O bus(es), or between a primary I/O bus and an attached (secondary) I/O bus. Bus bridges may be transparent to the software or may require more active participation of a bus bridge driver.
bus bridge driver	device driver software responsible for managing a bus bridge.
bus bridge metalanguage	a UDI metalanguage which covers registration and de-registration of interrupt handlers plus the delivery of bus-related events. It is (currently) subdivided into four interface sets, one for each of four roles: the bridge, the device, the interrupt handler and the interrupt dispatcher.
card-relative address	a physical memory address from the point of view of an adapter card's DMA engine. Such an address is used by the device to access main system memory, but may go through a translation process (in hardware) before reaching the memory.
DMA	Direct Memory Access by an I/O card to system memory without host processor intervention.
interrupt dispatcher	a driver which is responsible for managing interrupt control and routing hardware.
interrupt handler	a driver that is responsible for the interrupt-related, device-specific, software control and management of interrupt-associated hardware.
interrupt router	the entity, transparent to the interrupt handler driver, that steers an interrupt along a specific signal path between the interrupting device and an interrupt slot on a particular interrupt controller.

interrupt slot the index of an interrupt request source (starting from zero) relative to a particular interrupt controller. In UDI, the slot number is found in the device's instance attributes.

interrupt source a specific interrupt request line (or source ID, for messaged based interrupts) through which a device signals its asynchronous events. Depending on the interrupt routing, one or more interrupt sources may share the same interrupt slot. Multiple interrupt sources sharing the same interrupt slot is known as *shared interrupts*.

ISA 1) Instruction Set Architecture. Defines the binary machine language syntax and semantics for a particular type of processor or processor family.

2) Industry Standard Architecture. An I/O bus type originally designed for the IBM AT and used in many PCs.

PCI Peripheral Control Interconnect. A specific type of I/O bus.

PIO Programmed-I/O; i.e. transfer of data between the external device and processor memory under direct program control of the CPU. See DMA for contrast.

PMI Processor-Memory Interconnect: the bus or interconnect that links the host CPU with system memory and primary I/O bus bridges.

scatter-gather (scgth) element . a single address/length pair associated with a scatter/gather structure, that identifies one piece of data buffer memory which is contiguous with respect to card-relative addresses.

scatter-gather (scgth) segment an array of one or more scatter/gather elements. Each scatter/gather structure references one or more scatter/gather segments, as necessary to address the entire data buffer. The memory containing a scatter/gather segment is itself contiguous with respect to card-relative addresses.

scatter-gather (scgth) structure a driver-visible UDI structure that reflects the card-relative addresses of a possibly physically discontiguous but logically contiguous data buffer, which the driver uses to set up DMA transfers.

shared interrupts see *interrupt source*.

sync an environment service that synchronizes the hosts view of memory with an I/O card's view. This includes cache flushing and purging, flushing buffers in bus bridges, etc.

Index